Microsoft Visual Basic

Referencias:

MSDN Library Visual Studio 6.0.; Libro electrónico de ayuda de MSDN

Enclclopedia de Microsoft Visual Basic 6.0; Fco. Javier Ceballos; Alfaomega; 2000

¿Qué es Visual Basic?

La palabra "*Visual*" hace referencia al método que se utiliza para crear la interfaz gráfica de usuario (GUI). En lugar de escribir numerosas líneas de código para describir la apariencia y la ubicación de los elementos de la interfaz, simplemente puede agregar objetos prefabricados en su lugar dentro de la pantalla.

La palabra "*Basic*" hace referencia al lenguaje BASIC (Beginners All-Purpose Symbolic Instruction Code), Visual Basic ha evolucionado a partir del lenguaje BASIC original y ahora contiene centenares de instrucciones, funciones y palabras clave, muchas de las cuales están directamente relacionadas con la interfaz gráfica de Windows.

Microsoft Visual Basic, presenta una manera más rápida y sencilla de crear aplicaciones para Microsoft Windows®; proporciona un juego completo de herramientas que facilitan el desarrollo rápido de aplicaciones. El lenguaje de programación Visual Basic no es exclusivo de Visual Basic. La Edición para aplicaciones del sistema de programación de Visual Basic, incluida en Microsoft Excel, Microsoft Access y muchas otras aplicaciones Windows, utilizan el mismo lenguaje. El sistema de programación de Visual Basic, Scripting Edition (VBScript) es un lenguaje de secuencias de comandos ampliamente difundido y un subconjunto del lenguaje Visual Basic.

Sólo se necesitan unos minutos para crear una aplicación sencilla con Visual Basic.:

- **Puede crear la interfaz de usuario "dibujando" controles, como cuadros de texto y botones de comando, en un formulario.**
- **A continuación, establezca las propiedades del formulario y los controles para especificar valores como el título, el color y el tamaño.**
- **Finalmente, escriba el código para dar vida a la aplicación.**

Ventanas, eventos y mensajes

Pero para entender el proceso de desarrollo de una aplicación debe familiarizarse con el entorno Windows, el cual incluye tres conceptos clave: *ventanas, eventos y mensajes*.

Una **ventana** es una región rectangular con sus propios límites. Hay varios tipos de ventanas:

- De Explorador de Windows
- De documento dentro de su programa de proceso de textos
- Un cuadro de diálogo que emerge para recordarle una cita
- Controles como: un botón, los iconos, cuadros de texto, botones de opción, barras de menús.

ejemplos más comunes

Un **evento** es una acción reconocida por un formulario o un control. Los eventos pueden producirse mediante acciones del usuario, como hacer clic con el *mouse* (ratón) o presionar una tecla, mediante programación o incluso como resultado de acciones de otras ventanas.

Cada vez que se produce un evento se envía un **mensaje** al sistema operativo. El sistema procesa el mensaje y lo transmite a las demás ventanas. Entonces, cada ventana puede realizar la acción apropiada, basándose en sus propias instrucciones para tratar ese mensaje en particular (por ejemplo, volverse a dibujar cuando otra ventana la ha dejado al descubierto).

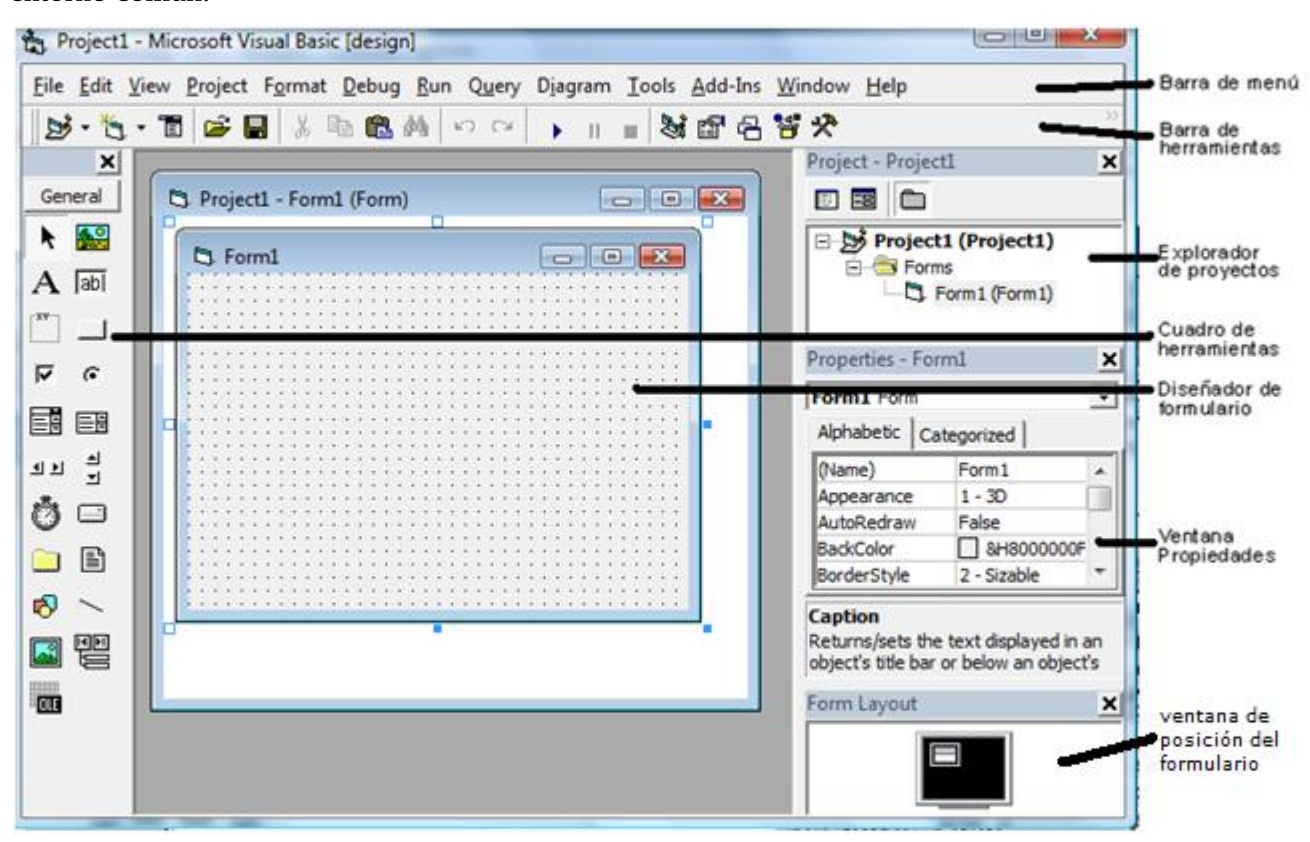
Visual Basic le evita tener que tratar con todos los controladores de mensajes de bajo nivel. Muchos de los mensajes los controla automáticamente Visual Basic, mientras que otros se

tratan como procedimientos de evento para su comodidad. Esto le permite crear rápidamente eficaces aplicaciones sin tener que tratar detalles innecesarios.

Desarrollo de aplicaciones

Entorno integrado de desarrollo (IDE)

Integra muchas funciones diferentes como el diseño, modificación, compilación y depuración en un entorno común.



Elementos del entorno integrado de desarrollo

El entorno integrado de desarrollo de Visual Basic (IDE) consta de los siguientes elementos:

Barra de menús. Presenta los comandos que se usan para trabajar con Visual Basic. Además de los menús estándar **Archivo (File)**, **Edición (Edit)**, **Ver (View)**, **Ventana (Window)** y **Ayuda (Help)**, se proporcionan otros menús para tener acceso a funciones específicas de programación como **Proyecto (project)**, **Formato (Format)**, **Depuración (Debug),** o **Correr (Run)**.

Menús contextuales. Contienen accesos directos a acciones que se realizan con frecuencia. Para abrir un menú contextual, haga clic con el botón secundario del *mouse* en el objeto que está usando. La lista específica de opciones disponibles en el menú contextual depende de la parte del entorno en la que se hace clic con el botón secundario del *mouse*.

Barras de herramientas. Proporcionan un rápido acceso a los comandos usados normalmente en el entorno de programación. Haga clic en un botón de la barra de herramientas para llevar a cabo la acción que representa ese botón. De forma predeterminada, al iniciar Visual Basic se presenta la barra de herramientas Estándar. Es posible activar o desactivar otras barras de herramientas adicionales para modificar, diseñar formularios desde el comando **Barras de herramientas** del menú **Ver (View)**. Las barras de herramientas se pueden acoplar debajo de la barra de menús o pueden "flotar" si selecciona la barra vertical del borde izquierdo y la arrastra fuera de la barra de menús.

Cuadro de herramientas. Proporciona un conjunto de herramientas que puede usar durante el diseño para colocar controles en un formulario.

Ventana Explorador de proyectos. Enumera los formularios y módulos del proyecto actual. Un *proyecto* es la colección de archivos que usa para generar una aplicación.

Ventana Propiedades. Enumera los valores de las propiedades del control o formulario seleccionado. Una *propiedad* es una característica de un objeto, como su tamaño, título o color.

Examinador de objetos. Enumera los objetos disponibles que puede usar en su proyecto y le proporciona una manera rápida de desplazarse a través del código. Puede usar el Examinador de objetos para explorar objetos en Visual Basic y otras aplicaciones, ver qué métodos y propiedades están disponibles para esos objetos, y pegar código de procedimientos en su aplicación.

Diseñador de formularios. Funciona como una ventana en la que se personaliza el diseño de la interfaz de su aplicación. Agregue controles, gráficos e imágenes a un formulario para crear la apariencia que desee. Cada formulario de la aplicación tiene su propia ventana diseñador de formulario.

Ventana Editor de código. Funciona como un editor para escribir el código de la aplicación. Se crea una ventana editor de código diferente para cada formulario o módulo del código de la aplicación.

Ventana Posición del formulario. La ventana Posición del formulario (figura 2.2) le permite colocar los formularios de su aplicación utilizando una pequeña representación gráfica de la pantalla.

Ventanas Inmediato, Locales e Inspección. Estas ventanas adicionales se proporcionan para la depuración de la aplicación. Sólo están disponibles cuando ejecuta la aplicación dentro del IDE.

Crear la interfaz de usuario

La interfaz de usuario es quizás la parte más importante de una aplicación; ciertamente, es la más visible. Para los usuarios, la interfaz es la aplicación; seguramente a ellos no les interesa el código que se ejecuta detrás. Independientemente del tiempo y el esfuerzo que haya empleado en la escritura y optimización del código, la facilidad de uso de su aplicación depende de la interfaz.

Cuando diseña una aplicación tiene que tomar muchas decisiones relacionadas con la interfaz. ¿Debe usar el estilo de documento único o el de documentos múltiples? ¿Cuántos formularios diferentes necesitará? ¿Qué comandos incluirá en los menús? ¿Usará barras de herramientas para

duplicar funciones de los menús? ¿Cómo va a hacer los cuadros de diálogo que interactúan con el usuario? ¿Qué nivel de asistencia necesita proporcionar?

Antes de empezar a diseñar la interfaz de usuario, tiene que pensar en el propósito de la aplicación. El diseño de una aplicación principal de uso constante debe ser diferente del de una que sólo se utiliza ocasionalmente, durante breves periodos de tiempo. Una aplicación cuyo propósito fundamental sea de presentar información tiene unos requisitos distintos que otra que se utilice para obtener información.

La audiencia prevista también debe influir en el diseño. Una aplicación destinada a usuarios principiantes requiere un diseño sencillo, mientras que una destinada a usuarios experimentados podría exigir uno más complejo. Otras aplicaciones utilizadas por los destinatarios del diseño pueden influir en la expectativa del comportamiento de su aplicación. Si piensa distribuir su producto internacionalmente, el idioma y la cultura de destino tienen que considerarse parte del diseño.

Al diseñar la interfaz de usuario de su aplicación debe tener en cuenta a los usuarios. ¿Con qué facilidad puede un usuario descubrir las características de su aplicación sin necesidad de entrenamiento? ¿Cómo responde la aplicación cuando se producen errores? ¿Qué ayuda o asistencia al usuario va a proporcionar? ¿El diseño le parece estético al usuario?

La mejor técnica de diseño de una interfaz de usuario es la del proceso iterativo; normalmente no se consigue un diseño perfecto a la primera. Este capítulo presenta el proceso de diseño de una interfaz en Visual Basic, proporcionando una introducción a las herramientas necesarias para crear excelentes aplicaciones para los usuarios.

Formularios, controles y menús

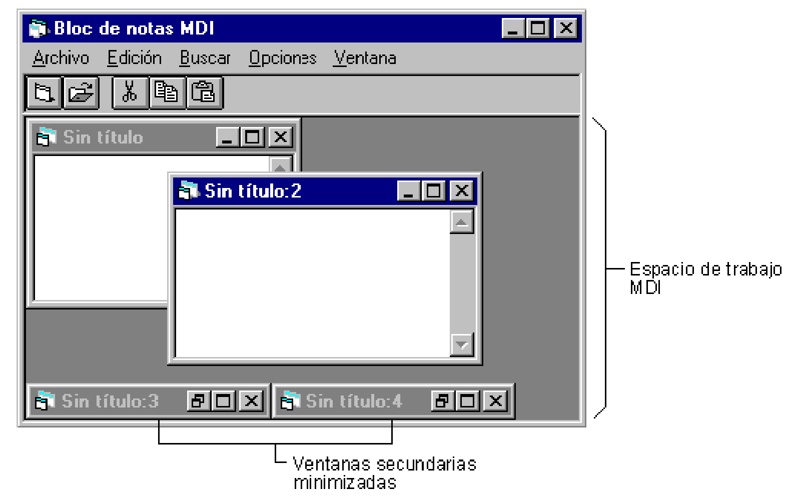
El primer paso para crear una aplicación con Visual Basic es crear la interfaz, la parte visual de la aplicación con la que va a interactuar el usuario. Los formularios y controles son los elementos de desarrollo básicos que se usan para crear la interfaz; son los objetos con los que se trabaja para desarrollar la aplicación.

Los formularios son objetos que exponen ***las propiedades que definen su apariencia (***se pueden considerar como atributos)***, los métodos que definen su comportamiento*** (se pueden considerar sus acciones) ***y los eventos que definen la forma en que interactúan con el usuario*** (se pueden considerar sus respuestas).

> *Mediante el establecimiento de las propiedades del formulario y la escritura de código de Visual Basic para responder a sus eventos se personaliza el objeto para cubrir las necesidades de la aplicación.*

Hay dos estilos principales de interfaz de usuario: la *interfaz de documento único* (*SDI*) y la *interfaz de documentos múltiples* (*MDI*). El estilo SDI es el más normal.

Un objeto **Form** es una ventana o un cuadro de diálogo que forma parte de la interfaz de usuario de una aplicación. **Forms** es una colección cuyos elementos representan cada formulario cargado en una aplicación. La colección incluye el formulario MDI, el formulario secundario MDI y formularios no MDI de la aplicación. La colección **Forms** tiene una única propiedad, **Count**, que especifica el número de elementos de la colección. El marcador de posición *índice* representa un entero entre 0 y `Forms.Count - 1`.



La interfaz de documentos múltiples (MDI) le permite crear una aplicación que mantenga varios formularios dentro de un único formulario contenedor. Los documentos o *ventanas secundarias* están contenidos en una *ventana primaria*, que proporciona un espacio de trabajo para todas las ventanas secundarias de la aplicación. En tiempo de ejecución, los formularios secundarios se presentan dentro del *espacio de trabajo* del formulario MDI primario (el área comprendida dentro de los bordes del formulario y bajo las barras de título y de menús). Cuando se minimiza un formulario secundario, su icono aparece dentro del espacio de trabajo del formulario MDI en lugar de aparecer en la barra de tareas. Aplicaciones como Microsoft Excel y Microsoft Word para Windows tienen interfaces de documentos múltiples.

Un formulario MDI se crea con el comando **Agregar formulario MDI** (Add MDIForm) del menú **Proyecto (Project)**; un formulario secundario MDI se crea eligiendo **Nuevo formulario (Add Form)** en el menú **Proyecto (Project)** y, a continuación, estableciendo la propiedad **MDIChild** a **True**.

Un formulario MDI es similar a un formulario normal con una restricción. No puede colocar un control directamente en un formulario MDI a menos que dicho control tenga una propiedad **Align** (como el control cuadro de imagen) o no tenga interfaz visible (como el control cronómetro).

En tiempo de ejecución, un formulario MDI y todos sus formularios secundarios tienen características especiales:

- Todos los formularios secundarios se presentan dentro del espacio de trabajo del formulario MDI. El usuario puede mover y ajustar el tamaño de los formularios secundarios como haría con cualquier otro formulario; sin embargo, están restringidos a dicho espacio de trabajo.
- Cuando se minimiza un formulario secundario, su icono aparece en el formulario MDI en lugar de aparecer en la barra de tareas. Cuando se minimiza el formulario MDI, el formulario MDI y todos sus formularios secundarios quedan representados por un único icono. Cuando se restaura el formulario MDI, el formulario MDI y todos sus formularios secundarios se presentan en el mismo estado en que se encontraban antes de minimizarse.
- Cuando se maximiza un formulario secundario, su título se combina con el del formulario MDI y se presenta en la barra de título del formulario MDI (vea la figura 6.6).
- Si establece la propiedad **AutoShowChildren** puede presentar los formularios secundarios automáticamente cuando se carguen los formularios (**True**) o puede cargar los formularios secundarios como ocultos (**False**).
- Los menús del formulario secundario activo (si los hubiera) se presentan en la barra de menús del formulario MDI, no en el formulario secundario.

Además de los dos estilos de interfaz más comunes, SDI y MDI, se está haciendo cada vez más popular un tercer estilo de interfaz: la interfaz al *estilo explorador*. La interfaz estilo explorador es una única ventana que contiene dos *paneles* o regiones, que normalmente consisten en una vista de

árbol o una vista jerárquica a la izquierda y un área de presentación a la derecha, como en el Explorador de Microsoft Windows. Este tipo de interfaz tiende por sí misma a la exploración o examen de grandes cantidades de documentos, imágenes o archivos.

Descripción de propiedades del formulario

Los formularios tienen propiedades que determinan aspectos de su apariencia (como posición, tamaño y color) y aspectos de su comportamiento (como si puede ajustar o no su tamaño).

El primer paso para diseñar un formulario consiste en establecer sus propiedades. Puede establecer las propiedades de un formulario en *tiempo de diseño* en la ventana Propiedades o en *tiempo de ejecución*, escribiendo código.

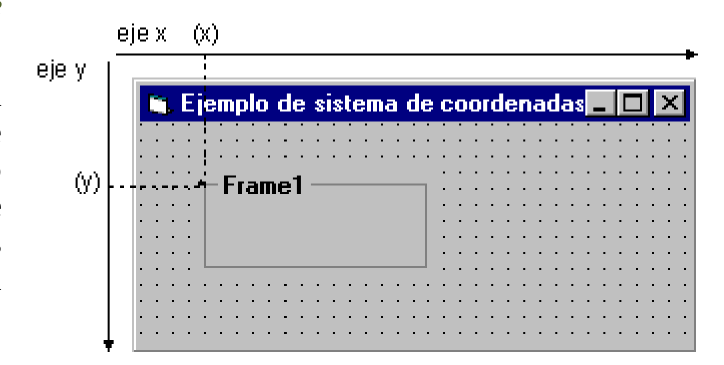
> **Nota:** *Tiempo de diseño*, que es cualquier momento mientras está desarrollando una aplicación en el entorno de Visual Basic, se trabaja con formularios y controles, se establecen propiedades y se escribe código para los eventos.
>
> *Tiempo de ejecución* es cualquier momento mientras se ejecuta realmente la aplicación y se interactúa con ella como lo haría un usuario.

Algunas propiedades de un formulario que afectan a su apariencia física:

- **Caption** determina el texto que muestra la barra de título del formulario
- **Icon** establece el icono que aparece en la esquina superior izquierda del formulario. (de 16x16 o de 32x32 pixeles, archivo con extensión .ico)
- **MaxButton** determinan si el formulario se puede maximizar.
- **MinButton** determinan si el formulario se puede minimizar.
- **ControlBox** determinan si el formulario se puede cerrar. Aparece el botón de cerrar.
    - Si desea que el formulario no tenga barra de titulo, cuadro del menú Control, botón Maximizar y botón Minimizar, elimine texto de la propiedad Caption y establezca a False las propiedades ControlBox, MaxButton y MinButton.
- **BorderStyle** para definir el borde del formulario.Si establece la propiedad en 0 se eliminará el borde.
- Las propiedades **Height** y **Width** determinan el tamaño inicial de un formulario,
- Las propiedades **Left** y **Top** determinan la ubicación del formulario en relación con la esquina superior izquierda de la pantalla.
- La propiedad **WindowState** puede establecer si el formulario se inicia en estado maximizado, minimizado o normal.
- La propiedad **Name** establece el nombre con el que hará referencia al formulario en el código. De forma predeterminada, cuando se agrega un formulario por primera vez a un proyecto, su nombre es Form1, Form2, etc. Es conveniente asignar a la propiedad **Name** un valor más significativo, como "frmEntry" para un formulario de entrada de pedidos.

Bloquear la posición de todos los controles en el formulario. Una vez que se haya ajustado el tamaño del formulario y de los controles y haya situado éstos en su posición definitiva, puede seleccionar el formulario y bloquear sus controles para que no puedan ser movidos accidentalmente. Ejecute ***Format/Locks Controls***. *Para desbloquear siga los mismos pasos.*

Descripción del sistema de coordenadas



El sistema de coordenadas es una cuadrícula bidimensional que define ubicaciones en la pantalla, en un formulario o en otro contenedor (como un cuadro de imagen o un objeto **Printer**). Las ubicaciones de esta cuadrícula se definen mediante las coordenadas del formulario:

(*x*, *y*)

El valor de *x* es la ubicación del punto sobre el eje x, con la ubicación predeterminada 0 en el extremo izquierdo. El valor de y es la ubicación del punto sobre el eje y, con la ubicación predeterminada 0 en el extremo superior.

Fig. El sistema de coordenadas de un formulario

El sistema de coordenadas de Visual Basic tiene las siguientes reglas:

- Cuando mueve o cambia el tamaño de un control, utiliza el sistema de coordenadas del contenedor del control. Si dibuja el objeto directamente en el formulario, el contenedor será el formulario. Si dibuja el control dentro de un marco o de un cuadro de imagen, el contenedor será el marco o el cuadro de imagen.

- Todos los métodos gráficos y el método **Print** utilizan el sistema de coordenadas del contenedor. Por ejemplo, las instrucciones que dibujan dentro de un cuadro de imagen utilizan el sistema de coordenadas del control **PictureBox**.

- Las instrucciones que cambian el tamaño o mueven un formulario siempre expresan la posición y el tamaño del formulario en twips[1].

  Cuando cree código para cambiar el tamaño de un formulario o para moverlo, antes debe comprobar las propiedades **Height** y **Width** del objeto **Screen** para asegurarse de que el formulario cabrá en la pantalla.

- La esquina superior izquierda de la pantalla siempre es (0, 0). El sistema de coordenadas predeterminado de cualquier contenedor empieza por la coordenada (0, 0) en la esquina superior izquierda del contenedor.

Las unidades de medida utilizadas para definir las ubicaciones a lo largo de los ejes se denominan colectivamente *escala*. En Visual Basic, cada eje del sistema de coordenadas puede tener su propia escala.

---

[1] Un *twip* es 1/20 de punto de impresora (1,440 twips equivalen a una pulgada y 567 twips equivalen a un centímetro). Estas medidas definen el tamaño de un objeto cuando se imprime. Las distancias físicas reales dentro de la pantalla varían de acuerdo con el tamaño de la pantalla. De forma predeterminada, todas las instrucciones de movimiento, cambio de tamaño y dibujo gráfico utilizan la unidad twip.

Puede modificar la dirección del eje, el punto de inicio y la escala del sistema de coordenadas, pero por ahora, utilice el sistema predeterminado.

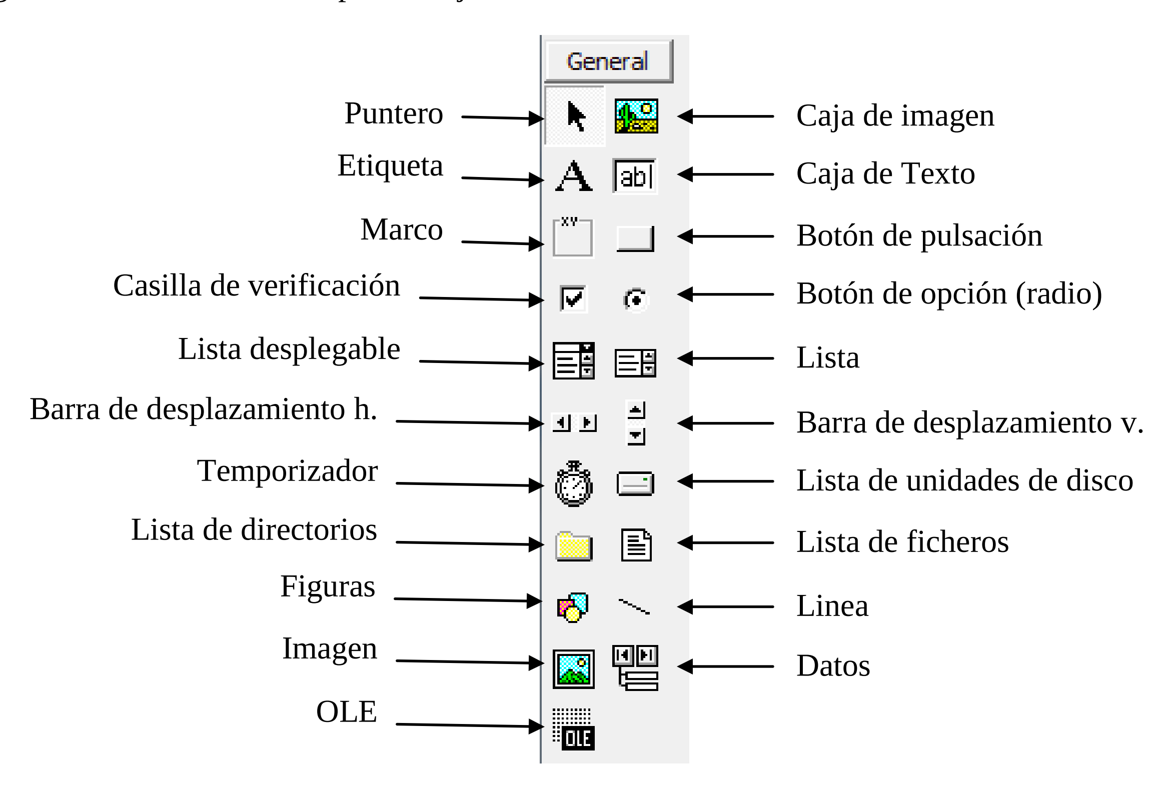
Controles

Los controles son objetos que están contenidos en los objetos de formularios. ***Cada tipo de control tiene su propio conjunto de propiedades, métodos y eventos***, que lo hacen adecuado para una finalidad determinada. Algunos de los controles que puede usar en las aplicaciones son más adecuados para escribir o mostrar texto, mientras que otros controles permiten tener acceso a otras aplicaciones y procesan los datos como si la aplicación remota formara parte del código.

Los objetos de un formulario son los elementos de desarrollo básicos de una aplicación de Visual Basic, las ventanas reales con las que interactúa el usuario cuando ejecuta la aplicación.

El cuadro de herramientas de Visual Basic contiene las herramientas que puede usar para dibujar controles en los formularios.

Fig. Cuadro de herramientas para dibujar controles



Hay tres categorías generales de controles en Visual Basic: *Controles intrínseco*, *Controles ActiveX*, *y los Objetos insertables*,

- *Controles intrínseco* como los controles de botón de comando y de marco. Estos controles se encuentran dentro del archivo ejecutable de Visual Basic. Los controles intrínsecos están siempre incluidos en el cuadro de herramientas, no como los controles ActiveX y los objetos insertables, que se pueden quitar o agregar al cuadro de herramientas.

- *Controles ActiveX*, que existen como archivos independientes con extensión de nombre de archivo .ocx. Entre éstos se incluyen los controles disponibles en todas las ediciones de Visual Basic (controles cuadrícula, cuadro combinado y lista enlazados a datos y varios más) y los que sólo están disponibles en la Edición Profesional y la Edición Empresarial (como **Listview**, **Toolbar**, **Animation** y **Tabbed Dialog**). También se encuentran disponibles muchos controles ActiveX de otros fabricantes.
- *Objetos insertables*, como un objeto **Worksheet** de Microsoft Excel que contiene la lista de los empleados de su empresa o un objeto **Calendar** de Microsoft Project que contiene la información del calendario de un proyecto. Puesto que estos objetos se pueden agregar al cuadro de herramientas, se les considera controles.

¿Qué es un objeto?

Un objeto es una combinación de código y datos que se puede tratar como una unidad. Un objeto puede ser una parte de una aplicación, como un control o un formulario. También puede ser un objeto una aplicación entera. La tabla siguiente describe ejemplos de tipos de objetos que puede usar en Visual Basic.

Ejemplo	**Descripción**
Botón de comando	Son objetos los controles de un formulario, como botones de comandos y marcos.
Formulario	Cada formulario de un proyecto de Visual Basic es un objeto distinto.
Base de datos	Las bases de datos son objetos y contienen otros objetos, como campos e índices.
Gráfico	Un gráfico de Microsoft Excel es un objeto.

¿De dónde vienen los objetos?

Cada objeto de Visual Basic se define mediante una *clase*. Para comprender la relación entre un objeto y su clase, piense en el molde de las galletas y las galletas. El molde es la clase. Define las características de cada galleta, como por ejemplo el tamaño y la forma. Se utiliza la clase para crear objetos. Los objetos son las galletas.

Mediante dos ejemplos se puede aclarar la relación que existe entre las clases y los objetos en Visual Basic.

- Los controles del cuadro de herramientas de Visual Basic representan clases. El objeto conocido como control no existe hasta que lo dibuja en un formulario. Cuando crea un control, está creando una copia o *instancia* de la clase del control. Esta instancia de la clase es el objeto al que hará referencia en la aplicación.
- El formulario en el que trabaja en tiempo de diseño es una clase. En tiempo de ejecución, Visual Basic crea una instancia de esa clase de formulario.

La ventana Propiedades muestra la clase y la propiedad **Name** de los objetos de una aplicación de Visual Basic.

Se crean todos los objetos como copias idénticas de sus clases. Una vez que existen como objetos individuales, es posible modificar sus propiedades. Por ejemplo, si dibuja tres botones de comando en un formulario, cada objeto botón de comando es una instancia de la clase CommandButton. Cada objeto comparte un conjunto de características y capacidades comunes (propiedades, métodos y

eventos), definidos por la clase. Sin embargo, cada uno tiene su propio nombre, se puede activar y desactivar por separado y se puede colocar en una ubicación distinta del formulario, etc.

Para simplificar, sólo recuerde que, por ejemplo, el término "control de cuadro de lista", significa "instancia de la clase ListBox".

¿Qué puede hacer con objetos?

Un objeto proporciona código que no tiene que escribir.

> Por ejemplo, puede crear su propios cuadros de diálogo **Abrir archivo** y **Guardar archivo**, pero no tiene por qué. En vez de eso, puede usar el control de diálogo común (un objeto) que Visual Basic proporciona. Podría escribir su propia agenda y código de administración de recursos, pero no tiene por qué. En su lugar, puede usar los objetos **Calendar**, **Resources** y **Tasks** que proporciona Microsoft Project.

Conceptos básicos del trabajo con objetos

Los objetos de Visual Basic aceptan propiedades, métodos y eventos. En Visual Basic, los datos de un objeto (configuración o atributos) se llaman propiedades, mientras que los diversos procedimientos que pueden operar sobre el objeto se conocen como sus métodos. Un evento es una acción reconocida por un objeto, como hacer clic con el *mouse* o presionar una tecla, y puede escribir código que responda a ese evento.

Puede cambiar las características de un objeto si modifica sus propiedades. Piense en una radio. Una propiedad de la radio es su volumen. En Visual Basic, se diría que una radio tiene la propiedad "Volumen" que se puede ajustar modificando su valor. Suponga que establece el volumen de la radio de 0 a 10. Si pudiera controlar una radio en Visual Basic, podría escribir el código en un procedimiento que modificara el valor de la propiedad "Volumen" de 3 a 5 para hacer que la radio suene más alto:

```
Radio.Volumen = 5
```

Además de propiedades, los objetos tienen métodos. Los métodos son parte de los objetos del mismo modo que las propiedades. Generalmente, los métodos son acciones que desea realizar, mientras que las propiedades son los atributos que puede establecer o recuperar. Por ejemplo, marca un número de teléfono para hacer una llamada. Se podría decir que el teléfono tiene un método "Marcar" y podría usar esta sintaxis para marcar el número de siete cifras 5551111:

```
Teléfono.Marcar 5551111
```

Los objetos también tienen eventos. Los eventos se desencadenan cuando cambia algún aspecto del objeto. Por ejemplo, una radio podría tener el evento "CambiarVolumen" y el teléfono podría tener el evento "Sonar".

Controles para mostrar e introducir texto

Los controles de etiquetas (control **Label**) y cuadros de texto (control **TextBox**) se usan para mostrar o introducir texto. Utilice etiquetas cuando desee que la aplicación muestre texto en un formulario y utilice cuadros de texto cuando desee permitir al usuario escribir texto. *Las etiquetas contienen texto que sólo se puede leer, mientras que los cuadros de texto contienen texto que se puede modificar.*

Control Label (Etiqueta)

Un control **Label** (etiqueta) muestra texto que el usuario no puede modificar directamente. *(Puede usar etiquetas para identificar controles que no tienen una propiedad **Caption**, como los cuadros de*

texto y las barras de desplazamiento.). Puede escribir código que cambie el texto mostrado por un control **Label** como respuesta a eventos en tiempo de ejecución.

Propiedades:

- **Caption** controla **e**l texto real que muestra una etiqueta. Se puede establecer en tiempo de diseño en la ventana Propiedades o en tiempo de ejecución si la asigna en el código. De forma predeterminada, el título es la única parte visible del control de etiqueta.
- **BackStyle** Devuelve o establece un valor que indica si un control **Label** o el fondo de un color **Shape** es transparente u opaco.
- **AutoSize** determina si se debe cambiar automáticamente el tamaño de un control para ajustarlo a su contenido. Si tiene el valor **True**, la etiqueta crece horizontalmente para ajustarse a su contenido.
- **WordWrap** hace que la etiqueta crezca verticalmente para ajustarse a su contenido, mientras conserva el mismo ancho.
- **Alignment** permite establecer la alineación del texto dentro del control a *Justificación izquierda* (0, el valor predeterminado), *Centrado* (1) o *Justificación derecha* (2)
- **UseMnemonic** estableciéndola a **True** permite definir un carácter en la propiedad **Caption** como tecla de acceso. Cuando define una tecla de acceso en un control **Label**, el usuario puede presionar y mantener presionado Alt+ el carácter que designe para mover el enfoque al control siguiente del orden de tabulación.

    También puede crear teclas de acceso directo para otros controles que tengan una propiedad **Caption** si inserta un carácter & delante de la letra que quiere usar como tecla de acceso.

    1. Dibuje primero la etiqueta y después dibuje el control. –o bien–

        Dibuje los controles en cualquier orden y establezca la propiedad **TabIndex** de la etiqueta a uno menos que la del control.

    2. Utilice un carácter & en la propiedad **Caption** de la etiqueta para asignar la tecla de acceso de la etiqueta.

        Nota Puede que desee presentar caracteres & en un control **Label**, en vez de usarlos para crear teclas de acceso. Esto puede ocurrir si enlaza un control **Label** con un campo de un conjunto de registros en el que los datos incluyen caracteres &. Para presentar caracteres & en un control **Label**, establezca la propiedad **UseMnemonic** a **False**.

Control TextBox (caja de texto)

Los cuadros de texto son controles versátiles que permiten obtener información del usuario o mostrar texto. No se deben usar cuadros de texto para mostrar texto que no desee que el usuario cambie.

Propiedades:

- **Locked** asignándole el valor **True** permite mostrar texto que no desee que el usuario cambie.
- **Text** muestra el texto real de un cuadro de texto. De forma predeterminada, un cuadro de texto presenta una única línea de texto y no muestra barras de desplazamiento. Si el texto es más largo que el espacio disponible, sólo será visible parte del texto. De forma predeterminada, puede

escribir en un cuadro de texto hasta 2048 caracteres. Si establece la propiedad **MultiLine** del control a **True**, puede escribir hasta 32 KB de texto.

Se puede establecer de tres formas diferentes:

- en tiempo de diseño en la ventana Propiedades
- en tiempo de ejecución si la establece en el código
- mediante el texto que escribe el usuario en tiempo de ejecución.

Se puede recuperar el contenido actual de un cuadro de texto si lee la propiedad **Text** en tiempo de ejecución.

- **MultiLine** asignándole el valor **True** activará un cuadro de texto para que acepte o muestre múltiples líneas de texto en tiempo de ejecución. Un cuadro de texto con múltiples líneas ajusta automáticamente el texto siempre y cuando no haya una barra de desplazamiento horizontal.
- **Alignment**. Cuando la propiedad **MultiLine** es **True**, también puede ajustar la alineación del texto a Justificación izquierda, Centrado o Justificación derecha. De forma predeterminada, el texto se justifica a la izquierda. Si la propiedad **MultiLine** es **False**, establecer la propiedad **Alignment** no tiene efecto.
- **ScrollBars** tiene el valor 0-None (ninguna barra) de forma predeterminada. El ajuste automático de línea ahorra al usuario la incomodidad de insertar saltos de línea al final de las líneas. Cuando una línea de texto es más larga de lo que se puede mostrar en una línea, el cuadro de texto ajusta automáticamente el texto a la línea siguiente.

No se pueden introducir saltos de línea en la ventana Propiedades en tiempo de diseño. Dentro de un procedimiento, para crear un salto de línea hay que insertar un retorno de carro seguido de un avance de línea (caracteres ANSI 13 y 10). También puede usar la constante **vbCrLf** para insertar una combinación de retorno de carro y avance de línea. Por ejemplo, el siguiente procedimiento de evento coloca dos líneas de texto en un cuadro de texto de múltiples líneas (Texto1) cuando se carga el formulario:

```
Sub Form_Load ()
   Texto1.Text = "Aquí hay dos líneas en" _
   & vbCrLf & "un cuadro de texto"
End Sub
```

- Puede controlar el punto de inserción y el comportamiento de la selección de un cuadro de texto mediante las propiedades **SelStart**, **SelLength** y **SelText**. Estas propiedades sólo están disponibles en tiempo de ejecución.

La primera vez que un cuadro de texto recibe el enfoque, el punto de inserción predeterminado, o posición del cursor, del cuadro de texto está a la izquierda de cualquier texto existente. El usuario puede moverlo desde el teclado o con el *mouse*. Si el cuadro de texto pierde el enfoque y lo vuelve a recuperar, el punto de inserción estará donde el usuario lo hubiera dejado la última vez.

En algunos casos este comportamiento puede ser desconcertante para el usuario. En una aplicación de procesamiento de textos, el usuario podría esperar que los caracteres nuevos aparecieran a continuación de cualquier texto existente. En una aplicación de entrada de datos, el usuario podría esperar que lo que escriba sustituya a cualquier dato existente. Las propiedades **SelStart** y **SelLength** permiten modificar el comportamiento para que cumpla una finalidad determinada.

- La propiedad **SelStart** es un número que indica que el punto de inserción está dentro de la cadena de texto, siendo 0 la posición situada más a la izquierda. Si la propiedad **SelStart** tiene un valor igual o mayor que el número de caracteres que hay en el cuadro de texto, el punto de inserción se situará después del último carácter.
- La propiedad **SelLength** es un valor numérico que establece el ancho del punto de inserción. Si asigna a **SelLength** un número mayor que 0 se seleccionarán y resaltarán ese número de caracteres a partir del punto de inserción actual.
- La propiedad **SelText** es una cadena de texto que puede asignar en tiempo de ejecución para reemplazar la selección actual. Si no hay texto seleccionado, **SelText** insertará su texto en el punto de inserción actual.

  Si el usuario comienza a escribir mientras está seleccionado un bloque de texto, se reemplazará el texto seleccionado. En algunos casos, quizás desee reemplazar una selección de texto con el texto nuevo mediante un comando de pegado.

- **PasswordChar** especifica el carácter presentado en el cuadro de texto. Por ejemplo, si quiere presentar asteriscos en el cuadro de contraseña, especifique * en la propiedad *PasswordChar* de la ventana Propiedades. Independientemente de los caracteres escritos por el usuario en el cuadro de texto, se presentan asteriscos.
- **MaxLength** determina cuántos caracteres se pueden escribir en el cuadro de texto. Cuando se sobrepasa **MaxLength**, el sistema emite un pitido y el cuadro de texto ya no acepta más caracteres.

Controles que muestran opciones a los usuarios

La mayoría de las aplicaciones necesitan presentar opciones a los usuarios, que van desde una simple opción de tipo sí o no hasta seleccionar de una lista que contiene cientos de posibilidades. Visual Basic incluye varios controles estándar que ayudan a presentar opciones. En la tabla siguiente se resumen estos controles y su uso adecuado.

Para proporcionar esta característica	Use este control	Grafico de ejemplo
Un conjunto pequeño de opciones entre las que el usuario puede elegir una o más.	**CheckBox** (casillas de verificación)	Negrita Cursiva Si está activada, Value = 1
Un conjunto pequeño de opciones entre las que el usuario sólo puede elegir una.	**OptionButton** (botones de opción; use marcos si son necesarios grupos adicionales)	486 Pentium Pentium Pro
Una lista desplegable de opciones entre las que puede elegir el usuario.	**ListBox** (cuadro de lista)	Paris Nueva York San Francisco Seattle Miami Los Angeles Chicago

Una lista desplegable de opciones junto con un cuadro de texto. El usuario puede elegir de la lista o escribir una opción en el cuadro de texto.	**ComboBox** (cuadro combinado	Lond París Nueva York San Francisco Seattle Miami Los Angeles Chicago Redmond

Casilla de verificación (Check Box)

Una casilla de verificación indica si una condición determinada está activada o desactivada. Se usan casillas de verificación en una aplicación para ofrecer a los usuarios opciones de tipo verdadero y falso o sí y no. Como las casillas de verificación funcionan independientemente una de otra, el usuario puede activar cualquier número de casillas de verificación al mismo tiempo.

Propiedades:

- **Caption** para establecer el texto que apareca junto al **CheckBox**.
- **Value** para determinar el estado del control: activado, desactivado o no disponible.

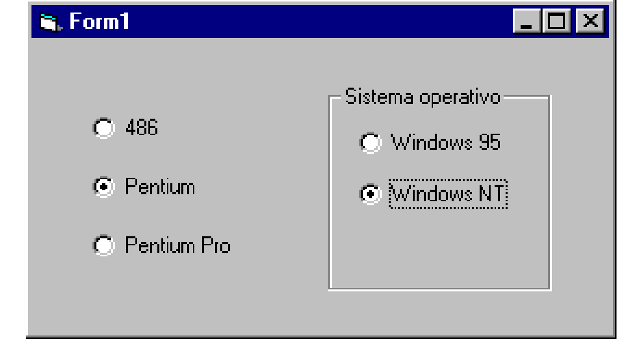
  Cuando activa una casilla la propiedad **Value** de la casilla de verificación es 1; cuando no está activada, su propiedad **Value** es 0. El valor predeterminado de **Value** es 0. Por eso, a menos que modifique **Value**, la casilla de verificación estará desactivada la primera vez que se muestre. Puede usar las constantes **vbChecked** y **vbUnchecked** para representar los valores 1 y 0.

  > De forma predeterminada, el control **CheckBox** se establece a **vbUnchecked**. Si quiere activar previamente varias casillas de verificación de una serie, puede hacerlo si establece sus propiedades **Value** a **vbChecked** en los procedimientos Form_Load o Form_Initialize.
  >
  > También puede establecer la propiedad **Value** a **vbGrayed** para deshabilitar la casilla de verificación. Por ejemplo, puede que desee deshabilitar una casilla de verificación cuando se den ciertas condiciones.

- **Style** al utilizarse permite mejorar la apariencia del la casilla de verificación Después utilice las propiedades **Picture**, **DownPicture** y **DisabledPicture**. Por ejemplo, puede que desee agregar un icono o un mapa de bits a una casilla de verificación o presentar imágenes diferentes cuando se hace clic en el control o cuando está deshabilitado.

Botones de opción (OptionButton)

Los botones de opción presentan al usuario un conjunto de dos o más opciones. Pero, a diferencia de las casillas de verificación, los botones de opción deben funcionar siempre como parte de un grupo; al activar un botón de opción se desactivan inmediatamente todos los demás botones del grupo. Al definir un botón de opción se indica al usuario "Aquí tiene un conjunto de opciones entre las que puede elegir una y sólo una".



Todos los botones de opción que se colocan directamente en un formulario (es decir, no en un marco o en un cuadro de imagen) constituyen un

grupo. Si desea crear grupos adicionales de botones de opción, debe colocarlos dentro de marcos o en cuadros de imagen.

Todos los botones de opción que haya dentro de un marco dado constituyen un grupo independiente, al igual que todos los botones de opción que haya dentro de un cuadro de imagen. Cuando cree un grupo independiente de esta forma, dibuje siempre primero el marco o el cuadro de imagen y, después, dibuje encima los botones de opción. En tiempo de diseño, los botones de opción contenidos en un control **Frame** o **PictureBox** se pueden seleccionar y mover como una única unidad.

Si bien los controles son objetos independientes, existe una cierta *relación primaria y secundaria* entre los formularios y los controles.

Para entender el concepto de contenedores, debe comprender que todos los controles son secundarios del formulario en el que se dibujan. De hecho, la mayoría de los controles admiten la propiedad de sólo lectura **Parent**, que devuelve el formulario en el que está ubicado un control. Ser secundario afecta a la colocación de un control en el formulario primario. Las propiedades **Left** y **Top** de un control son relativas al formulario primario y no se pueden mover los controles fuera de los límites del formulario primario. Mover un contenedor mueve también los controles, y la posición del control relativa a las propiedades **Left** y **Top** del contenedor no cambia, ya que el control se mueve con el contenedor.

Hay varias formas de seleccionar un botón de opción:

- Hacer clic en él con el *mouse* en tiempo de ejecución.
- Ir al grupo de botones de opción con el tabulador y usar luego las teclas de dirección para seleccionar un botón de opción del grupo.
- Asignar a la propiedad **Value** el valor **True** en el código:
```
optOpción.Value = True
```
- Usar una tecla de método abreviado especificada en el título de la etiqueta.

Propiedades

- **Value** asígnele el valor **True** en tiempo de diseño para hacer que un botón sea el predeterminado en un grupo de botones de opción. El botón permanecerá seleccionado hasta que un usuario seleccione otro botón de opción diferente o hasta que lo modifique el código.
- **Enabled** asigne el valor **False** para desactivar un botón de opción. Cuando se ejecute el programa aparecerá atenuado, lo que significa que no está disponible.

Cuadros de lista (**ListBox)** y los cuadros combinados (**ComboBox**)

Los cuadros de lista y los cuadros combinados son una manera efectiva de presentar al usuario gran cantidad de opciones en un espacio limitado, ya que presentan una lista de opciones. De forma predeterminada, las opciones se muestran verticalmente en una única columna, aunque también puede establecer múltiples columnas. Si el número de elementos supera a los que se pueden mostrar en el cuadro combinado o el cuadro de lista, aparecen automáticamente barras de desplazamiento en el control. El usuario puede entonces desplazarse por la lista hacia arriba o hacia abajo o de izquierda a derecha.

Un cuadro combinado aúna las características de un cuadro de texto y un cuadro de lista. Este control permite al usuario seleccionar opciones si escribe texto en el cuadro combinado o selecciona un elemento de la lista.

Los cuadros de lista y los cuadros combinados contienen múltiples valores o una colección de valores. Tienen métodos integrados para agregar, quitar y recuperar valores de sus colecciones en tiempo de ejecución. Para agregar varios elementos a un cuadro de lista llamado Lista1, el código sería como el siguiente:

```
Lista1.AddItem "París"
Lista1.AddItem "Nueva York"
Lista1.AddItem "San Francisco"
```

Puede insertar elementos en la lista en tiempo de diseño si establece la propiedad **List** en la ventana Propiedades del control **ListBox**. Cuando selecciona la propiedad **List** y después hace clic en la flecha hacia abajo, puede escribir un elemento de la lista y presionar la combinación de teclas Ctrl-Entrar para iniciar una línea nueva. Sólo puede agregar elementos al final de la lista. Por tanto, si quiere ordenar alfabéticamente la lista, establezca la propiedad **Sorted** a **True**.

Cuándo debe usar un cuadro combinado en lugar de un cuadro de lista

Generalmente, el cuadro combinado es el apropiado cuando hay una lista de opciones *sugeridas* y el cuadro de lista es el apropiado cuando desea limitar la entrada a las opciones de la lista. Un cuadro combinado contiene un campo de edición, de forma que en este campo se pueden introducir opciones que no figuran en la lista.

Además, los cuadros combinados ahorran espacio en los formularios. Como la lista completa no se presenta hasta que el usuario hace clic en la flecha abajo (excepto en el Estilo 1, donde siempre se presenta), un cuadro combinado puede ocupar un espacio reducido, en el que no cabría un cuadro de lista.

Propiedades de ListBox:

- La propiedad **Columns** permite especificar el número de columnas de un cuadro de lista. Esta propiedad puede tener los valores siguientes:

Valor	**Descripción**
0	Cuadro de lista de una única columna con desplazamiento vertical.
1	Cuadro de lista de una única columna con desplazamiento horizontal.
>1	Cuadro de lista de múltiples columnas con desplazamiento horizontal.

Visual Basic ajusta los elementos de la lista a la línea siguiente y agrega una barra de desplazamiento horizontal a la lista si es necesario; si la lista cabe en una única columna, no se agrega ninguna barra de desplazamiento. El ajuste a la siguiente columna también se produce automáticamente sólo si es necesario. Observe que si el ancho de una entrada de un cuadro de lista es mayor que el ancho de una columna, el texto se recorta.

- Puede permitir que los usuarios seleccionen múltiples elementos de una lista. La propiedad **MultiSelect** controla la selección múltiple en los cuadros de lista estándar. Dicha propiedad puede tener los valores siguientes:

Valor	Tipo de selección	Descripción
0	Ninguna	Cuadro de lista estándar.
1	Selección múltiple simple	Con un clic o con la tecla Barra espaciadora se seleccionan o se anulan la selección de elementos adicionales de la lista.
2	Selección múltiple extendida	Con las combinaciones Máyus-clic o Mayús- una tecla de dirección se extiende la selección para incluir todos los elementos que están entre la selección actual y la anterior. Ctrl-

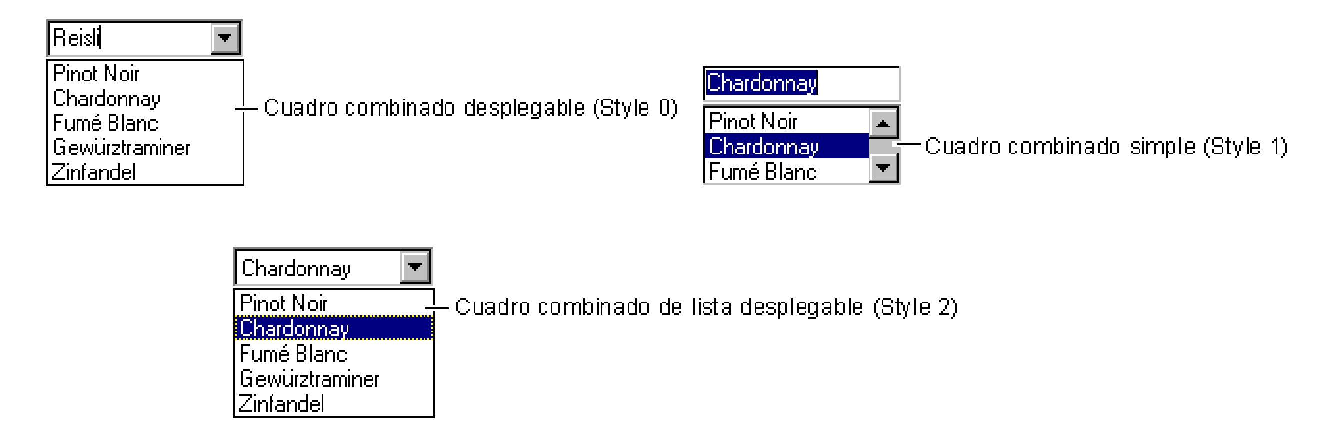
		clic selecciona o anula la selección de un elemento de la lista.

Propiedades de ComboBox:

- **Style.** Estilos de cuadros combinados. Hay tres estilos de cuadros combinados. Cada estilo puede establecerse en tiempo de diseño o en tiempo de ejecución y, para establecer el estilo del cuadro combinado, se utilizan valores o constantes equivalentes de Visual Basic.

Estilo	Valor	Constante
Cuadro combinado desplegable	0	vbComboDropDown
Cuadro combinado simple	1	vbComboSimple
Cuadro de lista desplegable	2	vbComboDropDownList

Fig. Los estilos de cuadro combinado



- **List** puede insertar elementos en la lista en tiempo de diseño. Cuando selecciona la propiedad y después hace clic en la flecha hacia abajo, puede escribir un elemento de la lista y presionar la combinación de teclas Ctrl-Entrar para iniciar una línea nueva.
- Normalmente, la forma más sencilla de obtener el valor del elemento seleccionado actualmente es mediante la propiedad **Text**. Esta propiedad se corresponde con el contenido de la parte de cuadro de texto del control en tiempo de ejecución. Puede ser un elemento de la lista o una cadena de texto escrita por el usuario en el cuadro de texto.

Propiedades comunes:

- Para obtener el número de elementos de un cuadro de lista, utilice la propiedad **ListCount**.
- Si quiere conocer la posición del elemento seleccionado de la lista, utilice la propiedad **ListIndex**.
- La propiedad **NewIndex** permite hacer un seguimiento del índice del último elemento agregado a la lista. Esto puede ser útil cuando inserta un elemento en una lista ordenada.
- Sólo puede agregar elementos al final de la lista. Por tanto, si quiere ordenar alfabéticamente la lista, establezca la propiedad **Sorted** a **True**. Puede especificar que los elementos se agreguen a la lista en orden alfabético si establece la propiedad **Sorted** a **True** y omite el

índice. El orden no distingue entre mayúsculas y minúsculas; por tanto, las palabras "chardonnay" y "Chardonnay" reciben el mismo tratamiento.

Barras de desplazamiento como dispositivos de entrada

Aunque las barras de desplazamiento suelen estar asociadas a cuadros de texto o a ventanas, algunas veces se usan como dispositivos de entrada de datos. Como estos controles pueden indicar la posición actual en una escala, se pueden usar individualmente los controles de barra de desplazamiento para controlar la entrada de datos en el programa; por ejemplo, para controlar el volumen de sonido o para ajustar los colores de una imagen.

Los controles **HScrollBar** (horizontal) y **VScrollBar** (vertical) funcionan independientemente de otros controles y tienen su propio conjunto de eventos, propiedades y métodos.

Los controles de barra de desplazamiento no son lo mismo que las barras de desplazamiento integradas que van adjuntas a los cuadros de texto, cuadros de lista, cuadros combinados o formularios MDI (los cuadros de texto y los formularios MDI tienen una propiedad **ScrollBars** para agregar o quitar barras de desplazamiento que están adjuntas al control).

Las instrucciones de diseño de la interfaz de Windows sugieren ahora usar como dispositivos de entrada controles deslizantes en lugar de barras de desplazamiento.

Controles que muestran imágenes y gráficos

Visual Basic incluye cuatro controles que facilitan el trabajo con gráficos: control de cuadro de imagen, control de imagen, control de forma y control de línea.

A veces se hace referencia a los controles de imagen, forma y línea como controles gráficos "ligeros". Requieren menos recursos del sistema y, por consiguiente, se muestran más deprisa que el control de cuadro de imagen; contienen un subconjunto de las propiedades, métodos y eventos que están disponibles en el cuadro de imagen. Cada uno de ellos es más adecuado para una finalidad determinada.

Para proporcionar esta característica	**Use este control**
Contenedor para otros controles	**Picture** (cuadro de imagen)
Métodos gráficos o de impresión.	**Picture** (cuadro de imagen)
Mostrar una imagen.	**Image** (control de imagen) o **Picture** (cuadro de imagen)
Mostrar un elemento gráfico simple	**Shape** (control de forma) o **Line** (control de línea)

Control de cuadro de imagen (PictureBox)

La utilización principal del control de cuadro de imagen es mostrar una imagen al usuario. El control **PictureBox** se utiliza para presentar gráficos, para actuar como contenedor de otros controles y para presentar el resultado de los métodos gráficos o texto con el método **Print**. El control **PictureBox** es similar al control **Image** en que los dos se pueden usar para presentar gráficos en una aplicación y los dos aceptan los mimos formatos gráficos. Sin embargo, el control **PictureBox** tiene funcionalidades que no tiene el control Image; por ejemplo, la posibilidad de actuar como contenedor de otros controles y de aceptar métodos gráficos.

El control **PictureBox** puede presentar archivos de imágenes en los formatos siguientes: mapa de bits, icono, metarchivo, metarchivo mejorado o archivos JPEG o GIF.

Propiedades

- **Picture** permite mostrar la imagen, contiene el nombre de archivo (y la ruta de acceso opcional) para el archivo de imagen que desea mostrar.

  Nota Los objetos **Form** (de formulario) tienen también una propiedad **Picture** que se puede establecer para mostrar una imagen directamente sobre el fondo del formulario.

  Para presentar o reemplazar una imagen *en tiempo de ejecución*, puede usar la función **LoadPicture** para establecer la propiedad **Picture**. Debe proporcionar el nombre (y una ruta de acceso opcional) para la imagen. La función **LoadPicture** controla los detalles de cómo cargar y mostrar la imagen:

```
      picPrincipal.Picture = LoadPicture("VANGOGH.BMP")
o indicando toda la ruta
```

```
      Set Picture1.Picture = _
      LoadPicture("c:\Windows\Winlogo.cur", vbLPLarge, vbLPColor)
```

- **AutoSize** cuando tiene el valor **True**, hace que el cuadro de imagen cambie de tamaño automáticamente para coincidir con las dimensiones de su contenido.

  Hay que tener mucho cuidado al diseñar el formulario si piensa usar un cuadro de imagen con la propiedad **AutoSize** activada. La imagen cambiará de tamaño sin tener en cuenta los demás controles del formulario, causando posiblemente resultados inesperados, como tapar otros controles. Es conveniente probarlo; para ello, cargue cada una de las imágenes en tiempo de diseño.

El control de cuadro de imagen puede usarse también como contenedor para otros controles. Como en el control de marco, puede dibujar otros controles encima del cuadro de imagen. Los controles contenidos se mueven con el cuadro de imagen y sus propiedades **Top** y **Left** serán relativas al cuadro de imagen en lugar de ser relativas al formulario.

Una utilización común para el contenedor del cuadro de imagen es como barra de herramientas o como barra de estado. Puede colocar controles de imagen en él para que actúen como botones o agregar etiquetas para presentar mensajes de estado. Si asigna a la propiedad **Align** el valor **Top**, **Bottom**, **Left** o **Right**, el cuadro de imagen se "pegará" al borde del formulario.

El control de cuadro de imagen tiene varios métodos que lo hacen útil para otras finalidades. El cuadro de imagen puede considerarse un lienzo en blanco sobre el que se puede pintar, dibujar o imprimir. Es posible usar un único control para mostrar texto, gráficos o incluso una simple animación.

El método **Print** permite escribir texto en el control de cuadro de imagen igual que se escribe en una impresora. Hay varias propiedades de fuentes disponibles para controlar las características del texto por medio del método **Print**; se puede usar el método **Cls** para borrar lo escrito.

Se pueden usar los métodos **Circle**, **Line**, **Point** y **Pset** para dibujar gráficos en el cuadro de imagen. Hay propiedades como **DrawWidth**, **FillColor** y **FillStyle** que permiten personalizar la apariencia de los gráficos.

Es posible crear animación con el método **PaintPicture**, si mueve imágenes dentro del control de imagen y cambia rápidamente entre varias imágenes diferentes.

Controles gráficos ligeros

Los controles de imagen (**Image)**, forma (**Shape)** y línea (**Line)** se consideran controles ligeros; es decir, sólo admiten un subconjunto de las propiedades, métodos y eventos que se encuentran en el cuadro de imagen. Por tanto, normalmente requieren menos recursos del sistema y se cargan más rápidamente que el control de cuadro de imagen.

El **control de imagen** es similar al control de cuadro de imagen, pero sólo se usa para mostrar imágenes. No tiene capacidad para actuar como contenedor de otros controles ni admite los métodos avanzados del cuadro de imagen.

Las imágenes se cargan en el control de imagen igual que en el cuadro de imagen: en tiempo de diseño, asignando a la propiedad **Picture** un nombre y una ruta de acceso de archivo, y en tiempo de ejecución mediante la función **LoadPicture**.

El comportamiento de cambio de tamaño del control de imagen difiere del comportamiento del cuadro de imagen. Tiene una propiedad **Stretch**, mientras que el cuadro de imagen tiene una propiedad **AutoSize**. Asignar a la propiedad **AutoSize** el valor **True** hace que un cuadro de imagen se ajuste a las dimensiones de la imagen, mientras que si se le asigna **False** hace que la imagen se recorte (sólo es visible una parte de la imagen). Cuando tiene el valor **False** (el valor predeterminado), la propiedad **Stretch** del control de imagen hace que se ajuste a las dimensiones de la imagen. Si la propiedad **Stretch** es **True**, la imagen se ajustará al tamaño del control de imagen, lo que puede hacer que la imagen aparezca distorsionada.

Se puede usar un control de imagen para crear sus propios botones, ya que reconoce también el evento Click. Por tanto, puede usarlo en cualquier sitio donde usaría un botón de comando. Es una forma cómoda de crear un botón con una imagen en lugar de un título. Agrupar varios controles de imagen horizontalmente en la parte superior de la pantalla (normalmente dentro de un cuadro de imagen) permite crear una barra de herramientas en la aplicación.

Para crear un borde alrededor del control de imagen, asigne a la propiedad **BorderStyle** el valor 1-Fixed Single.

> **Nota** A diferencia de los botones de comando, los controles de imagen no aparecen presionados cuando se hace clic en ellos. Esto significa que, a menos que modifique el mapa de bits del evento MouseDown, no hay ninguna indicación visual para el usuario de que el "botón" está presionado.

Usar controles de forma y de línea

Los controles de forma y de línea son útiles para dibujar elementos gráficos en la superficie de un formulario. *Estos controles no admiten eventos; son estrictamente para fines decorativos.*

Hay varias propiedades para controlar la apariencia del *control de forma*:

- **Shape** si se establece, se puede presentar como un rectángulo, cuadrado, óvalo, círculo, rectángulo redondeado o cuadrado redondeado.
- **BorderStyle** para especificar el estilo de la línea en tiempo de diseño. Proporciona seis estilos de línea: Transparente(0), Continua, Guión, Punto, Guión-punto, Guión-punto-punto, Continua interna .
- **BorderWidth** Devuelve o establece el ancho del borde de un control.
    - Puede usar las propiedades **BorderWidth** y **BorderStyle** para especificar el tipo de borde que desea en un control **Line** o **Shape**.
- **BackColor** devuelve o establece el color de la línea o el color de fondo de un objeto.

- **BorderColor** devuelve o establece el color del borde de un objeto.
- **FillColor** Devuelve o establece el color usado para llenar formas.
    - Cuando **BorderStyle** es 0 (Transparente), se pasa por alto la propiedad **BorderColor**.
- **FillStyle** Devuelve o establece el patrón usado para llenar controles **Shape**, así como los círculos y los cuadros creados con los métodos gráficos **Circle** y **Line**.

Constante	**Valor**	**Descripción**
vbFSSolid	0	Sólido.
vbFSTransparent	1	(Predeterminado) Transparente.
vbHorizontalLine	2	Línea horizontal.
vbVerticalLine	3	Línea vertical.
vbUpwardDiagonal	4	Diagonal hacia arriba.
vbDownwardDiagonal	5	Diagonal hacia abajo.
vbCross	6	Cruz.
vbDiagonalCross	7	Cruz diagonal.

Cuando la propiedad **FillStyle** se establece a su valor predeterminado, 1 (Transparente), el valor de **FillColor** se pasa por alto, excepto en el objeto **Form**.

- **DrawMode** Devuelve o establece un valor que determina la apariencia del resultado de un método gráfico o la apariencia de un control **Shape** o **Line**. Use esta propiedad para producir efectos visuales con controles **Shape** o **Line** o al dibujar con métodos gráficos. Visual Basic compara cada píxel del patrón de dibujo con el píxel correspondiente del fondo existente y después aplica operaciones en el nivel de bit.

Constante	**Valor**	**Descripción**
vbBlackness	1	Negro.
vbNotMergePen	2	Inverso de combinar el lapiz y la muestra: inverso del valor 15.
vbMaskNotPen	3	Combinar fondo e inverso del lápiz: combinación de los colores comunes en el color de fondo y el inverso de la pluma.
vbNotCopyPen	4	Inverso de copiar del lápiz: inverso del valor 13.
vbMaskPenNot	5	Combinar el lápiz y el inverso de la muestra: combinación de los colores comunes a la pluma y al inverso de la presentación.
vbInvert	6	Invertir: inverso del color de presentación.
vbXorPen	7	Operación Xor con el lápiz: combinación de los colores de la pluma y del color de presentación, pero no de ambos.
vbNotMaskPen	8	Inverso de combinar el lápiz y los colores comunes: inverso del valor 9.
vbMaskPen	9	Combinación del lápiz y la presentación: combinación de los colores comunes a la pluma y a la presentación.
vbNotXorPen	10	Inverso de la operación Xor con el lápiz: inverso del valor 7.
vbNop	11	Ninguna operación. El resultado permanece sin cambios. De hecho, este valor desactiva el dibujo.
vbMergeNotPen	12	Combinación de la muestra y el inverso del lápiz: combinación del color de presentación y el inverso del color de la pluma.

vbCopyPen	13	Copiar del lápiz (predeterminado): color especificado por la propiedad **ForeColor**.
vbMergePenNot	14	Combinación del lápiz y el inverso de la muestra: combinación del color de la pluma y el inverso del color de presentación.
vbMergePen	15	Combinación del lápiz y la muestra: combinación del color de la pluma y el color de presentación.
vbWhiteness	16	Blanco.

El control de línea es similar al control de forma, pero sólo se puede usar para dibujar líneas rectas.

Boton de comando o de pulsación (CommandButton)

Utilice un control **CommandButton** para comenzar, interrumpir o terminar un proceso. Un usuario siempre puede elegir un **CommandButton** si hace clic en él. Cuando se hace clic en él, invoca el comando escrito en su procedimiento de evento Click. La mayoría de las aplicaciones de Visual Basic tienen botones de comando que permiten que el usuario simplemente haga clic en ellos para realizar ciertas acciones. Cuando el usuario hace clic en el botón, no sólo ejecuta éste la acción apropiada; también parece como si lo presionara y lo soltara, por lo que algunas veces se llama botón pulsador.

Propiedades:

- **Caption** para mostrar texto en un control **CommandButton**, puede escribir hasta 255 caracteres. Puede usar la propiedad **Caption** para crear teclas de acceso directo en los botones de comando si inserta un carácter & delante de la letra que quiere usar como tecla de acceso directo. Por ejemplo, para crear una tecla de acceso para el título "Imprimir", agregaría un carácter & delante de la letra "I": "&Imprimir".
- Puede cambiar la fuente de presentación del botón de comando si establece la propiedad **Font**.
- Puede modificar el tamaño de los botones de comando con el *mouse* (ratón) o si establece las propiedades **Height** y **Width**.
- **Default** establecida a **True** para especificar un botón de comando como el predeterminado, permite que el usuario lo elija presionando la tecla ENTRAR, incluso aunque cambie el enfoque a un control diferente de un botón de comando. En cada formulario, puede hacer que un botón de comando sea el botón de comando predeterminado.
- **Cancel** establecida a **True** sirve para establecer un botón de cancelación predeterminado, permitir que el usuario elija el botón presionando la tecla ESC, incluso aunque cambie el enfoque a otro control.
- **Value** se establece a **True** y se desencadena su evento Click. El valor **False** (predeterminado) indica que el botón no está seleccionado. Puede usar la propiedad **Value** en el código para desencadenar el evento Click de un botón de comando.
- Puede mejorar la apariencia del control **CommandButton**, al igual que la de los controles **CheckBox** y **OptionButton**, si altera el valor de la propiedad **Style** y utiliza después las propiedades **Picture**, **DownPicture** y **DisabledPicture**. Por ejemplo, puede que desee agregar un icono o un mapa de bits a un botón de comando o presentar una imagen diferente cuando se hace clic en el control o cuando está deshabilitado.

Control Marco (Frame)

Los controles Frame se utilizan para proporcionar una agrupación identificable para otros controles. Por ejemplo, los controles Frame se pueden usar para subdividir un formulario funcionalmente, es decir, para separar grupos de controles de botón de opción.

En la mayoría de los casos, el control Frame se utiliza de forma pasiva (para agrupar otros controles) y no necesita responder a sus eventos. Sin embargo, probablemente se cambiarán sus propiedades **Name**, **Caption** o **Font**.

Para añadir otros controles al marco, dibújelos dentro de él. Si dibuja un control fuera del marco, o utiliza el método del doble clic para agregar un control a un formulario y, a continuación, intenta moverlo dentro del control **Frame**, el control estará encima del marco y tendrá que mover el marco y los controles independientemente. Para seleccionar múltiples controles en un **Frame**, mantenga presionada la tecla CTRL mientras utiliza el *mouse* (ratón) para dibujar un cuadro alrededor de los controles.

Control temporizador (Timer)

Los controles **Timer** responden al paso del tiempo. Son independientes del usuario y puede programarlos para que ejecuten acciones a intervalos periódicos de tiempo. Un uso típico es comprobar la hora del sistema para ver si es el momento de ejecutar alguna tarea. Los cronómetros también son útiles para otros tipos de procesamiento en segundo plano.

Los controles **Timer** tienen una propiedad **Interval** que especifica el número de milisegundos transcurridos entre un evento del cronómetro y el siguiente. A menos que esté desactivado, un control **Timer** sigue recibiendo un evento (llamado evento Timer) a intervalos iguales de tiempo.

La propiedad **Interval** presenta algunas limitaciones que debe tener en cuenta al programar un control **Timer**:

- Si cualquier aplicación carga mucho al sistema, por ejemplo con bucles muy largos, cálculos intensivos o acceso a unidades, a la red o a puertos, puede que una aplicación no reciba los eventos Timer con la frecuencia especificada en la propiedad **Interval**.

- El intervalo puede estar entre 0 y 64.767, ambos inclusive, lo que significa que incluso el mayor intervalo no puede ser mucho mayor de un minuto (unos 64,8 segundos).

- No se garantiza que el intervalo tenga siempre la misma duración. Para asegurar la precisión, el cronómetro debe comprobar la hora del sistema cuando sea necesario, en lugar de mantener el tiempo acumulado de forma interna.

- El sistema genera 18 pasos de reloj por segundo, de forma que aunque la propiedad **Interval** se mide en milisegundos, la precisión real del intervalo no es mayor que una decimoctava parte de segundo.

Los controles **Timer** deben estar asociados a un formulario. Por tanto, para crear una aplicación de cronómetro, debe crear al menos un formulario (aunque no tiene que hacerlo visible si no lo necesita para algún otro fin).

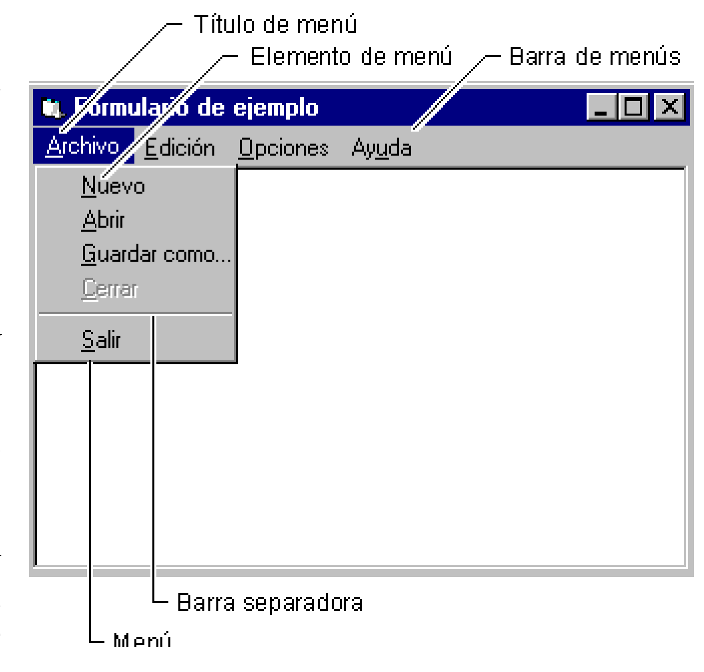
El control **Timer** tiene dos propiedades principales:

- **Enabled** Si quiere que el cronómetro empiece a funcionar en cuanto se carga el formulario, establezca esta propiedad a **True**. De lo contrario, deje la propiedad establecida a **False**. Puede que desee que otro evento externo (como hacer clic en un botón de comando) inicie el funcionamiento del cronómetro.
  - Observe que la propiedad **Enabled** del cronómetro es diferente de la propiedad **Enabled** de otros objetos. En la mayoría de los objetos, la propiedad **Enabled** determina si el objeto puede responder a eventos provocados por el usuario. En el control **Timer**, cuando se establece **Enabled** a **False** se suspende el funcionamiento del cronómetro.
- **Interval** Número de milisegundos transcurridos entre los eventos Timer.
  - Recuerde que el evento Timer es *periódico*. La propiedad **Interval** no determina la duración sino la frecuencia. La duración del intervalo depende de la precisión que desee. Como existe cierta posibilidad de errores, haga que la precisión del intervalo sea el doble de la deseada.

Nota: Cuando mayor es la frecuencia del evento Timer, más tiempo del procesador se utiliza en las respuestas al evento. Esto puede afectar al rendimiento en general. No defina intervalos demasiado cortos a menos que sea realmente necesario.

Fundamentos de los menús

Si desea que la aplicación proporcione un conjunto de comandos a los usuarios, los menús ofrecen una forma cómoda y coherente de agrupar comandos y una manera sencilla de que los usuarios tengan acceso a ellos.

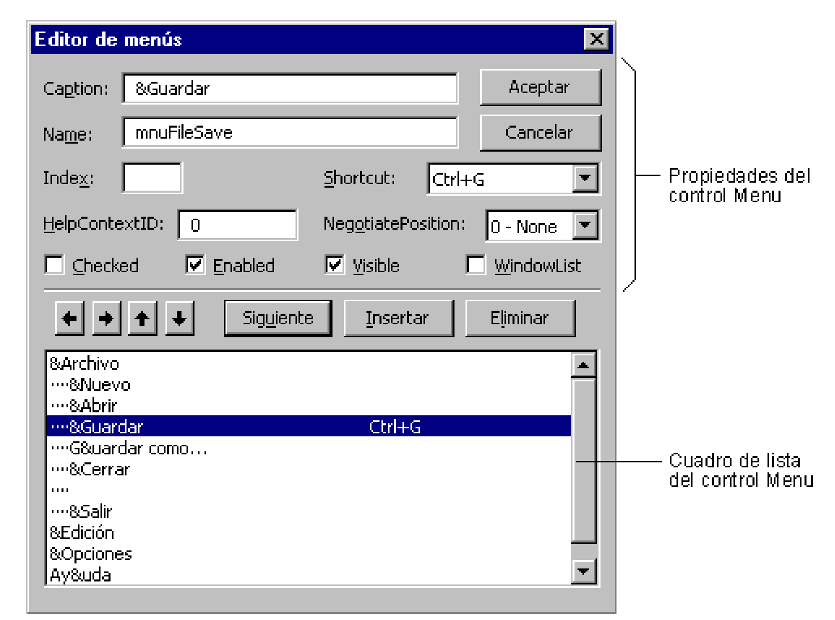


La *barra de menús* aparece en el formulario inmediatamente debajo de la *barra de título* y contiene uno o más *títulos de menús*. Cuando hace clic en un título de menú (como **Archivo**), se despliega un menú que contiene una lista de elementos de menú. Los elementos de menú pueden incluir comandos (como **Nuevo** y **Salir**), barras de separación y títulos de submenús. Cada elemento de menú que ve el usuario corresponde a un control de menú definido en el Editor de menús

Para que la aplicación sea más fácil de usar, debe agrupar los comandos de menú según su función.

- Algunos elementos de menú realizan una acción directamente; por ejemplo, el elemento **Salir** del menú **Archivo** cierra la aplicación.
- Otros elementos de menú muestran un *cuadro de diálogo (una ventana que requiere que el usuario proporcione información que la aplicación necesita para realizar la acción).* Estos elementos de menú deben ir seguidos de puntos suspensivos (…). Por ejemplo, cuando elige **Guardar como...** en el menú **Archivo** aparece el cuadro de diálogo **Guardar como**.

Un **control de menú** es un objeto. Como otros objetos, tiene propiedades que puede usar para definir su apariencia y su comportamiento. Puede establecer la propiedad **Caption**, las propiedades **Enabled** y **Visible**, la propiedad **Checked** y otras, en tiempo de diseño o en tiempo de ejecución. *Los controles de menú contienen sólo un evento, el evento Click*, que se invoca al seleccionar el control de menú con el *mouse* o mediante el teclado.



Usar el Editor de menús

Con el Editor de menús es posible agregar comandos nuevos a menús existentes, reemplazar comandos de menú existentes con sus propios comandos, crear nuevos menús y barras de menús, y modificar y eliminar menús y barras de menús existentes. La principal ventaja del Editor de menús es su facilidad de uso. Puede personalizar los menús de una manera completamente interactiva que conlleva muy poca programación.

Para presentar el Editor de menús en el menú **Herramientas (Tools)**, elija **Editor de menús (Menu Editor)**.

Aunque la mayoría de las propiedades del control de menú pueden establecerse mediante el Editor de menús, todas las propiedades de menú están también disponibles en la ventana Propiedades. Normalmente se crean los menús en el Editor de menús, pero para modificar rápidamente una propiedad puede usar la ventana Propiedades.

Propiedades comunes a algunos controles

- **BorderStyle** si se le asigna el valor 1 (*lo que puede hacer en tiempo de diseño)*, la etiqueta aparece con un borde que le da una apariencia similar a un cuadro de texto.
- **BackColor** devuelve o establece el color de fondo de un objeto
- **ForeColor** devuelve o establece el color de primer plano utilizado para mostrar texto y gráficos en un objeto
- **Enabled** Devuelve o establece un valor que determina si un formulario o un control puede responder a eventos generados por el usuario, como eventos del teclado y del *mouse*.
- **Font** Devuelve o establece un objeto **Font**. El objeto **Font** contiene la información necesaria para dar formato al texto que se presenta en la interfaz de una aplicación o que se envía a una impresora.

- **Index** Devuelve o establece el número que identifica un control de forma exclusiva en una matriz de controles[2]. Sólo está disponible si el control forma parte de una matriz de controles.
- **Name** establece el nombre con el que hará referencia al control en el código.
- **Visible** Devuelve o establece un valor que indica si un objeto es visible o está oculto.

Antes de empezar a codificar algunas convenciones de codificación

Es importante establecer convenciones de nomenclatura. De forma predeterminada, Visual Basic asigna el nombre Form1 al primer formulario de un proyecto, al segundo Form2 y así sucesivamente. Si tiene muchos formularios en una aplicación, es conveniente darles nombres significativos para evitar confusiones cuando escriba o modifique el código.

La razón principal de usar un conjunto coherente de convenciones de código es estandarizar la estructura y el estilo de codificación de una aplicación de forma que el autor y otras personas puedan leer y entender el código fácilmente. El objetivo es hacer que el programa sea fácil de leer y de entender sin obstruir la creatividad natural del programador con imposiciones excesivas y restricciones arbitrarias.

Prefijos sugeridos para controles

Tipo de control	Prefijo	Ejemplo
Botones de comando	cmd	**cmd**Salir
Etiqueta	lbl	**lbl**MensajeAyuda
Cuadro de texto	txt	**txt**Apellido
Cuadro de lista	lst	**lst**Codigos
Cuadro combinado, cuadro de lista desplegable	cbo	**cbo**Ingles
Botón de opción	opt	**opt**Genero
Casilla de verificación	chk	**chk**SoloLectura

Prefijos sugeridos para menús

Los prefijos de controles de menús se deben extender más allá de la etiqueta inicial "mnu", agregando un prefijo adicional para cada nivel de anidamiento, con el título del menú final en la última posición de cada nombre. En la tabla siguiente hay algunos ejemplos.

Secuencia del título del menú	Nombre del controlador del menú
Archivo	mnuArchivo
Archivo / Abrir	mnuArchivoAbrir
Archivo / Enviar correo	mnuArchivoEnviarCorreo
Archivo / Enviar fax	mnuArchivoEnviarFax
Formato	mnuFormato

[2] Grupo de controles que comparten un nombre, tipo y procedimientos de evento comunes. Cada control en una matriz tiene un número de índice único que se puede utilizar para determinar qué control reconoce un evento.

Formato / Carácter	mnuFormatoCaracter
Ayuda	mnuAyuda
Ayuda / Contenido	mnuAyudaContenido

Cuando se usa esta convención de nombres, todos los miembros de un grupo de menús determinado se muestran uno junto a otro en la ventana Propiedades de Visual Basic. Además, los nombres del control de menú documentan claramente los elementos de menú a los que están adjuntos

Nombres descriptivos de variables, constantes y procedimientos

El cuerpo de un nombre de variable o procedimiento se debe escribir en mayúsculas y minúsculas y debe tener la longitud necesaria para describir su funcionalidad. *Además, los nombres de funciones deben empezar con un verbo, como IniciarNombreMatriz o CerrarDiálogo.*

Cuando se usen abreviaturas, hay que asegurarse de que sean coherentes en toda la aplicación. Alternar aleatoriamente entre *Cnt* y *Contar* dentro de un proyecto provoca una confusión innecesaria.

Cuando escribe código en Visual Basic, declara y asigna nombre a muchos elementos (procedimientos **Sub** y **Function**, variables, constantes, etc.). Los nombres de procedimientos, variables y constantes que declara en el código de Visual Basic deben seguir estas directrices:

- Deben comenzar por una letra.
- No pueden contener puntos o caracteres de declaración de tipos (caracteres especiales que especifican tipos de datos.
- No pueden superar los 255 caracteres. Los nombres de controles, formularios, clases y módulos no deben exceder los 40 caracteres. Para nombres que se usen con frecuencia o para términos largos, se recomienda usar abreviaturas estándar para que los nombres tengan una longitud razonable. En general, los nombres de variables con más de 32 caracteres pueden ser difíciles de leer en pantallas angostas.
- No pueden ser iguales que las palabras clave restringidas.

Una *palabra clave restringida* es una palabra que Visual Basic utiliza como parte de su lenguaje. Esto incluye a las instrucciones predefinidas (como **If** y **Loop**), funciones (como **Len** y **Abs**) y operadores (como **Or** y **Mod**). Lista de palabras clave:

As	Binary	Public	Error	False	Date
Else	ByRef	Step	Input	Is	For
Friend	ByVal	True	Me	Mid	Len
Let	Empty	ParamArray	Null	On	New
Next	Get	Resume	Print	Private	Option
Optional	Lock	String	Seek	Set	Property
To	Nothing	WithEvents	Then	Time	Static

GoSub	Const	Declare	Dim	Function	Name
Unlock	Open	ReDim	Sub	Type	Call
GoTo	End	If	Select	Case	Each

Los formularios y controles pueden tener el mismo nombre que una palabra clave restringida. Por ejemplo, puede tener un control que se llame Loop. Sin embargo, en el código no puede hacer

referencia a ese control de la forma habitual, ya que Visual Basic supone que se está refiriendo a la palabra clave **Loop**. Por ejemplo, este código provoca un error:

```
Loop.Visible = True            ' Provoca un error.
```

Para referirse a un formulario o un control que tenga el mismo nombre que una palabra clave restringida, debe calificarlo o ponerlo entre corchetes: [ ]. Por ejemplo, este código no provoca un error:

```
MiForm.Loop.Visible = True    ' Calificado con el nombre del formulario
[Loop].Visible = True          ' Con corchetes también funciona.
```

Puede usar corchetes de esta forma cuando haga referencia a formularios y controles, pero no cuando declare una variable o defina un procedimiento con el mismo nombre que una palabra clave restringida. También se pueden usar corchetes para que Visual Basic acepte nombres de otras bibliotecas de tipos que entren en conflicto con palabras clave restringidas.

Nota Puesto que escribir corchetes puede resultar tedioso, evite la utilización de palabras clave restringidas en los nombres de formularios y controles. Sin embargo, puede usar esta técnica si una versión posterior de Visual Basic define una palabra clave nueva que entre en conflicto con un nombre de formulario o control existente y esté actualizando el código para que funcione con la versión nueva.

Variables

Declarar todas las variables ahorra tiempo de programación porque reduce el número de errores debidos a erratas (por ejemplo, aNombreUsuarioTmp frente a sNombreUsuarioTmp frente a sNombreUsuarioTemp).

Las variables deben llevar un prefijo para indicar su tipo de datos. Opcionalmente, y en especial para programas largos, el prefijo se puede ampliar para indicar el alcance de la variable.

Tipos de datos de variables

Use los prefijos siguientes para indicar el tipo de datos de una variable.

Tipo de datos	**Prefijo**	**Ejemplo**
Boolean	bln	**bln**Encontrado
Byte	byt	**byt**DatosImagen
Objeto Collection	col	**col**Widgets
Currency	cur	**cur**Ingresos
Date (Time)	dtm	**dtm**Inicio
Double	dbl	**dbl**Tolerancia
Error	err	**err**NumDeOrden
Integer	int	i**nt**Cantidad
Long	lng	**lng**Distancia
Object	obj	**obj**Activo
Single	sng	**sng**Media
String	str	**str**NombreF
Tipo definido por el usuario	udt	**udt**Empleado
Variant	vnt	**vnt**CheckSum

Prefijos de alcance de variables

A medida que aumenta el tamaño del proyecto, también aumenta la utilidad de reconocer rápidamente el alcance de las variables. Esto se consigue escribiendo un prefijo de alcance de una letra delante del tipo de prefijo, sin aumentar demasiado la longitud del nombre de las variables.

Alcance	**Prefijo**	**Ejemplo**
Global	g	**g***str*NombreUsuario
Nivel de módulo	m	**m***bln*ProgresoDelCalculo
Local del procedimiento	*(Ninguno prefijo)*	dblVelocidad

Constantes

El cuerpo del nombre de las constantes se debe escribir en mayúsculas. Aunque las constantes estándar de Visual Basic no incluyen información de tipo de datos y el alcance, los prefijos como *i*, *s*, *g* y *m* pueden ser muy útiles para entender el valor y el alcance de una constante. Para los nombres de constantes, se deben seguir las mismas normas que para las variables. Por ejemplo:

```
MINTMAXLISTAUSUARIO      ' Límite de entradas máximas para la lista de
usuarios
                         ' (valor entero, local del módulo)
GSTRNUEVALINEA           ' Carácter de nueva línea cadena, global de la
                         'aplicación)
```

Convenciones de codificación estructurada

Además de las convenciones de nombres, las convenciones de codificación estructurada, como comentarios al código y sangrías coherentes, pueden mejorar mucho la legibilidad del código.

Convenciones de comentarios al código

Todos los procedimientos y funciones deben comenzar con un comentario breve que describa las características funcionales del procedimiento (qué hace). Esta descripción no debe describir los detalles de implementación (cómo lo hace), porque a veces cambian con el tiempo, dando como resultado un trabajo innecesario de mantenimiento de los comentarios o, lo que es peor, comentarios erróneos. El propio código y los comentarios de líneas necesarios describirán la implementación.

Los argumentos que se pasan a un procedimiento se deben describir cuando sus funciones no sean obvias y cuando el procedimiento espera que los argumentos estén en un intervalo específico. También hay que describir, al principio de cada procedimiento, los valores devueltos de funciones y las variables globales que modifica el procedimiento, en especial los modificados a través de argumentos de referencia.

Los bloques del comentario de encabezado del procedimiento deben incluir los siguientes encabezados de sección.

Encabezado de sección	**Descripción del comentario**
Finalidad	Lo que hace el procedimiento (no cómo lo hace).
Premisas	Lista de cada variable externa, control, archivo abierto o cualquier otro elemento que no sea obvio.
Efectos	Lista de cada variable externa, control o archivos afectados y el efecto que tiene (sólo si no es obvio).
Entradas	Todos los argumentos que puedan no ser obvios. Los argumentos se

	escriben en una línea aparte con comentarios de línea.
Resultados	Explicación de los valores devueltos por las funciones.

Recuerde los puntos siguientes:

- Cada declaración de variable importante debe incluir un comentario de línea que describa el uso de la variable que se está declarando.
- Las variables, controles y procedimientos deben tener un nombre bastante claro para que los comentarios de línea sólo sean necesarios en los detalles de implementación complejos.
- Al principio del módulo .bas que contiene las declaraciones de constantes genéricas de Visual Basic del proyecto, debe incluir un resumen que describa la aplicación, enumerando los principales objetos de datos, procedimientos, algoritmos, cuadros de diálogo, bases de datos y dependencias del sistema. Algunas veces puede ser útil un pseudocódigo que describa el algoritmo.

Dar formato al código

Hay que ajustarse al espacio de la pantalla en la medida de lo posible y hacer que el formato del código siga reflejando la estructura lógica y el anidamiento. Estos son algunos indicadores:

- Los bloques anidados estándar, separados por tabuladores, deben llevar una sangría de cuatro espacios (predeterminado).
- El comentario del esquema funcional de un procedimiento debe llevar una sangría de un espacio. Las instrucciones de nivel superior que siguen al comentario del esquema deben llevar una sangría de un tabulador, con cada bloque anidado separado por una sangría de un tabulador adicional. Por ejemplo:

```
'*****************************************************
' Finalidad:   Ubica el primer caso encontrado de un
'               usuario especificado en la matriz
               ListaUsuario.
' Entradas:
'   strListaUsuario():   lista de usuarios para buscar.
'   strUsuarioDest:   nombre del usuario buscado.
' Resultados:   Índice del primer caso de rsUsuarioDest
'             encontrado en la matriz rasListaUsuario.
'             Si no se encuentra el usuario de destino,
'             devuelve -1.
'*****************************************************

Function intFindUser (strUserList() As String, strTargetUsuer As _
   String)As Integer
   Dim i As Integer            ' Contador de bucle.
   Dim blnFound As Integer   ' Indicador de
                           ' destino encontrado.
   intFindUser = -1
   i = 0
   While i <= Ubound(strUserList) and Not blnFound
      If strUserList(i) = strTargetUser Then
         blnFound = True
         intFindUser = i
      End If
   Wend
End Function
```

Agrupación de constantes

Las variables y constantes definidas se deben agrupar por funciones en lugar de dividirlas en áreas aisladas o archivos especiales. Las constantes genéricas de Visual Basic se deben agrupar en un único módulo para separarlas de las declaraciones específicas de la aplicación.

Operadores & y +

Use siempre el operador **&** para unir cadenas y el operador **+** cuando trabaje con valores numéricos. El uso del operador **+** para concatenar puede causar problemas cuando se opera sobre dos variables **Variant**. Por ejemplo:

```
vntVar1 = "10,01"
vntVar2 = 11
vntResult = vntVar1 + vntVar2    'vntResult = 21,01
vntResult = vntVar1 & vntVar2   'vntResult = 10,0111
```

Crear cadenas para MsgBox, InputBox y consultas SQL

Cuando esté creando una cadena larga, use el carácter de subrayado de continuación de línea para crear múltiples líneas de código, de forma que pueda leer o depurar la cadena fácilmente. Esta técnica es especialmente útil cuando se muestra un cuadro de mensaje (**MsgBox**) o un cuadro de entrada (**InputBox**), o cuando se crea una cadena SQL. Por ejemplo:

```
Dim Msg As String
Msg = "Esto es un párrafo que estará en un" _
& " cuadro de mensajes. El texto está separado en" _
& " varias líneas de código en el código fuente, " _
& "lo que facilita al programador la tarea de leer y depurar."
MsgBox Msg

Dim CTA As String
CTA = "SELECT *" _
& " FROM Título" _
& " WHERE [Fecha de publicación] > 1988"
ConsultaTítulos.SQL = CTA
```

Fundamentos de programación

Tras crear la interfaz de la aplicación mediante formularios y controles, necesitará escribir el código que defina el comportamiento de la aplicación. Como cualquier lenguaje moderno de programación, Visual Basic acepta ciertas construcciones de programación y elementos de lenguaje comunes.

Visual Basic es un lenguaje de programación basado en objetos.

Visual Basic interpreta el código a medida que lo escribe, interceptando y resaltando la mayoría de los errores de sintaxis en el momento. Es casi como tener un experto vigilando cómo escribe el código. A diferencia de los lenguajes tradicionales, Visual Basic utiliza una aproximación interactiva para el desarrollo, difuminando la distinción entre los tres pasos (escritura, compilación y comprobación del código.)

Para interceptar errores sobre la marcha, Visual Basic también compila parcialmente el código según se escribe. Cuando esté preparado para ejecutar y probar la aplicación, tardará poco tiempo en terminar la compilación. Si el compilador encuentra un error, quedará resaltado en el código. Puede corregir el error y seguir compilando sin tener que comenzar de nuevo.

A causa de la naturaleza interactiva de Visual Basic, se encontrará ejecutando la aplicación frecuentemente a medida que la desarrolle. De esta forma puede probar los efectos del código según lo escriba en lugar de esperar a compilarlo más tarde.

Definir el formulario inicial

De forma predeterminada, el primer formulario de la aplicación es el *formulario inicial*. Cuando la aplicación inicia la ejecución, se presenta este formulario (el primer código que se ejecuta es el del evento ***Form_Initialize*** de dicho formulario). Si quiere presentar un formulario diferente cuando se inicie la aplicación, debe cambiar el formulario inicial.

Para cambiar el formulario inicial

1. En el menú **Proyecto**, elija **Propiedades del proyecto**.
2. Elija la ficha **General**.
3. En el cuadro de lista **Objeto inicial**, seleccione el formulario que desee que sea el nuevo formulario inicial.
4. Elija **Aceptar**.

Inicio sin formulario inicial

Algunas veces puede desear que la aplicación se inicie sin cargar ningún formulario. Por ejemplo, puede que desee ejecutar código que cargue un archivo de datos y después presentar uno de entre varios formularios, según el contenido de dicho archivo. Puede hacerlo creando un procedimiento **Sub** llamado Main en un módulo estándar, como en el siguiente ejemplo:

```
Sub Main()
   Dim intStatus As Integer
   ' Llamar a un procedimiento de función para comprobar el estado
   ' del usuario.
```

```
    intStatus = GetUserStatus
    ' Mostrar un formulario inicial distinto según el estado.
    If intStatus = 1 Then
       frmMain.Show
    Else
       frmPassword.Show
    End If
End Sub
```

Este procedimiento tiene que ser un procedimiento **Sub** y no puede estar en un módulo de formulario. Para establecer el procedimiento **Sub Main** como objeto inicial, en el menú **Proyecto** elija **Propiedades del proyecto**, seleccione la ficha **General** y seleccione **Sub Main** en el cuadro **Objeto inicial**.

Presentar una pantalla de espera al iniciar

Si necesita ejecutar un procedimiento de larga duración al iniciar, como cargar una gran cantidad de datos de una base de datos o cargar varios mapas de bits grandes, puede que desee presentar una pantalla de espera al iniciar. Una pantalla de espera es un formulario que normalmente presenta información como el nombre de la aplicación, información de la propiedad intelectual y un mapa de bits. La pantalla que aparece cuando se inicia Visual Basic es una pantalla de espera.

Para presentar una pantalla de espera, use un procedimiento **Sub Main** como objeto inicial y utilice el método **Show** para presentar el formulario:

```
Private Sub Main()
    frmSplash.Show              ' Mostrar la pantalla de espera.
    ' Agregue aquí sus procedimientos de inicio.
    …
    ' Mostrar el formulario principal y descargar la pantalla de espera.
    frmMain.Show
    Unload frmSplash
End Sub
```

La pantalla de espera llama la atención del usuario mientras se ejecutan las rutinas de inicio, con lo que se da la impresión de que la aplicación se está cargando más rápidamente. Cuando terminan las rutinas de inicio, puede cargar el primer formulario y descargar la pantalla de espera.

Cuando diseñe una pantalla de espera, procure que sea sencilla. Si utiliza mapas de bits grandes o muchos controles, la propia pantalla de espera tardaría en cargarse.

Terminar una aplicación

Una aplicación controlada por eventos se termina cuando se cierran todos sus formularios y no se ejecuta ningún código. Si un formulario se mantiene oculto cuando se cierre el último formulario visible, parecerá que la aplicación ha terminado (porque no hay formularios visibles), pero en realidad se seguirá ejecutando hasta que se cierren todos los formularios ocultos. Esta situación puede producirse porque el acceso a las propiedades o controles de un formulario no cargado hace que éste se cargue de forma implícita, sin presentarlo.

La mejor manera de evitar este problema cuando se cierre la aplicación es asegurarse de descargar todos los formularios. Si tiene más de un formulario, puede usar la colección **Forms** y la instrucción **Unload**. Por ejemplo, en el formulario principal puede tener un botón de comando llamado cmdQuit que permita al usuario salir del programa. Si la aplicación sólo tiene un formulario, el procedimiento de evento Click puede ser tan sencillo como éste:

```
Private Sub cmdQuit_Click ()
   Unload Me
End Sub
```

Si la aplicación utiliza varios formularios, puede descargar los formularios agregando código en el procedimiento de evento Unload del formulario principal. Puede usar la colección **Forms** para asegurarse de buscar y cerrar todos los formularios. El siguiente código usa la colección **Forms** para descargar todos los formularios:

```
Private Sub Form_Unload (Cancel As Integer)
   Dim i as Integer
   ' Recorre la colección Forms y descarga
   ' cada formulario.
   For i = Forms.Count - 1 to 0 Step - 1
      Unload Forms(i)
   Next
End Sub
```

Puede haber casos en los que necesite terminar su aplicación sin que importe el estado de los formularios u objetos existentes. Para este propósito Visual Basic proporciona la instrucción **End**.

La instrucción **End** termina una aplicación de forma inmediata: después de la instrucción **End** no se ejecuta nada de código y no se producen más eventos. En concreto, Visual Basic no ejecuta los procedimientos de evento QueryUnload, Unload o Terminate de ningún formulario. Se liberan las referencias a los objetos, pero si ha definido sus propias clases, Visual Basic no ejecutará los eventos Terminate de los objetos creados a partir de las clases.

Además de la instrucción **End**, la instrucción **Stop** también detiene una aplicación. Sin embargo, la instrucción **Stop** sólo debe usarse en la fase de depuración, ya que no libera las referencias a los objetos.

Estructura de una aplicación en Visual Basic

La estructura de una aplicación es la forma en que se organizan las instrucciones; es decir, dónde se almacenan las instrucciones y el orden en que se ejecutan. La organización no es muy importante cuando sólo se tiene una línea de código. A medida que las aplicaciones se van haciendo más complejas, resulta obvia la necesidad de organizar o estructurar.

Puesto que una aplicación de Visual Basic se basa en objetos, la estructura de su código se aproxima mucho a su representación física en pantalla. Por definición, los objetos contienen datos y código.

El código de una aplicación de Visual Basic está organizado de forma jerárquica. Una aplicación típica está compuesta por uno o más módulos. Cada módulo contiene uno o más procedimientos que contienen código. Determinar qué procedimientos pertenecen a cada módulo dependerá del tipo de aplicación que esté creando. Como Visual Basic está basado en objetos, esto le ayudará a pensar en la aplicación en función de los objetos que representa.

Hay tres clases de módulos: *formularios (.frm), módulos estándar (.bas) y clases (.cls)*. Cada modulo puede contener:

- Declaraciones de constantes, de tipos, de variables y de procedimientos de DDL.
- Procedimientos conducidos por eventos
- Procedimientos y funciones generales (estándar)

Módulos de formulario

Los módulos de formulario (extensión de nombre de archivo .frm) son la base de la mayoría de las aplicaciones de Visual Basic. Por cada formulario de una aplicación hay un *módulo de formulario* relacionado (con la extensión de nombre de archivo .frm) que contiene su código. Pueden contener procedimientos que controlen eventos, procedimientos generales y declaraciones a nivel de formulario de variables, constantes, tipos y procedimientos externos.

La base de una aplicación en Visual Basic la forman sus procedimientos conducidos por eventos. Cada módulo de formulario contiene *procedimientos de evento* (secciones de código donde se colocan las instrucciones que se ejecutarán como respuesta a eventos específicos). Un procedimiento conducido por un evento es el código que es invocado cuando un objeto reconoce que ha ocurrido un determinado evento.

Los formularios pueden contener controles. Por cada control de un formulario, existe el correspondiente conjunto de procedimientos de evento en el módulo de formulario.

Si examina un módulo de formulario con un editor de textos, podrá ver las descripciones del formulario y sus controles, así como los valores de sus propiedades. El código que se escribe en un módulo de formulario es específico de la aplicación a la que pertenece el formulario y puede hacer referencia a otros formularios u objetos de la aplicación.

Módulos estándar

Además de procedimientos de evento, los módulos de formulario pueden contener procedimientos generales que se ejecutan como respuesta a una llamada desde cualquier procedimiento de evento. Los módulos estándar (extensión de nombre de archivo .bas) son contenedores de los procedimientos y declaraciones a los que tienen acceso otros módulos de la aplicación.

Cuando varios procedimientos conducidos por eventos necesiten ejecutar un mismo proceso, por ejemplo visualizar un diagrama de barras, la mejor forma de proceder es colocar el código común en un *procedimiento estándar*, perteneciente a un módulo estándar(.bas), que será invocado desde cada procedimiento conducido por un evento que necesite ejecutar dicho código. De esta forma se elimina la necesidad de duplicar código. Un procedimiento estándar es invocado cuando se hace una llamada explicita al mismo.

Pueden contener declaraciones globales (disponibles para toda la aplicación) o a nivel de módulo de variables, constantes, tipos, procedimientos externos y procedimientos globales. El código que se escribe en un módulo estándar no está ligado necesariamente a una aplicación determinada; si tiene cuidado de no hacer referencia a controles o formularios por su nombre, puede reusar un módulo estándar en distintas aplicaciones.

Un procedimiento estándar puede escribirse como procedimiento **Sub** o como función **Function**. Un procedimiento conducido por un evento siempre es un procedimiento **Sub.**

De forma predeterminada, el proyecto contiene un único módulo de formulario.

Descripción del modelo controlado por eventos

En una aplicación controlada por eventos, el código no sigue una ruta predeterminada; ejecuta distintas secciones de código como respuesta a los eventos. Los eventos pueden desencadenarse por

acciones del usuario, por mensajes del sistema o de otras aplicaciones, o incluso por la propia aplicación. La secuencia de estos eventos determina la secuencia en la que se ejecuta el código, por lo que la ruta a través del código de la aplicación es diferente cada vez que se ejecuta el programa.

> En las aplicaciones tradicionales o "por procedimientos", la aplicación es la que controla qué partes de código y en qué secuencia se ejecutan. La ejecución comienza con la primera línea de código y continúa con una ruta predefinida a través de la aplicación, llamando a los procedimientos según se necesiten.

Cómo funciona una aplicación controlada por eventos

Las aplicaciones controladas por eventos ejecutan código Basic como respuesta a un evento. Cada formulario y control de Visual Basic tiene un conjunto de eventos predefinidos. Si se produce uno de dichos eventos y el procedimiento de evento asociado tiene código, Visual Basic llama a ese código.

> Aunque los objetos de Visual Basic reconocen automáticamente un conjunto predefinido de eventos, usted decide cuándo y cómo se responderá a un evento determinado. A cada evento le corresponde una sección de código (un procedimiento de evento). Cuando desea que un control responda a un evento, escribe código en el procedimiento de ese evento.

Los tipos de eventos reconocidos por un objeto varían, pero muchos tipos son comunes a la mayoría de los controles.

> Por ejemplo, la mayoría de los objetos reconocen el evento Click: si un usuario hace clic en un formulario, se ejecuta el código del procedimiento de evento Click del formulario; si un usuario hace clic en un botón de comando, se ejecuta el código del procedimiento de evento Click del botón. El código en cada caso será diferente.

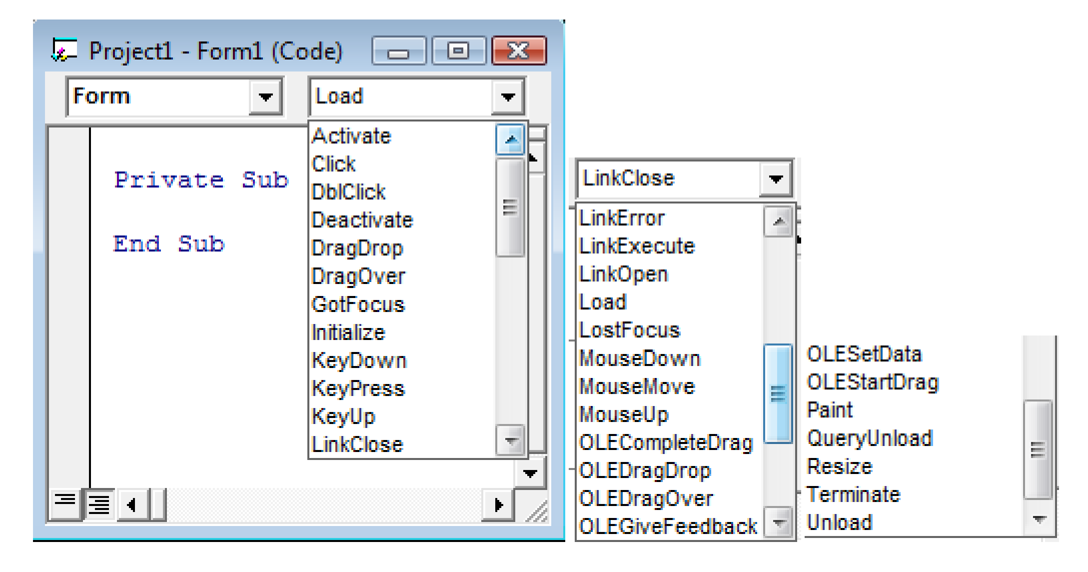
He aquí una secuencia típica de eventos en una aplicación controlada por eventos:

1. Se inicia la aplicación y se carga y muestra un formulario.
2. El formulario (o un control del formulario) recibe un evento. El evento puede estar causado por :
   - el usuario (por ejemplo, por la pulsación de una tecla),
   - por el sistema (por ejemplo, un evento de cronómetro) o,
   - de forma indirecta, por el código (por ejemplo, un evento Load cuando el código carga un formulario).
3. Si hay código en el procedimiento de evento correspondiente, se ejecuta.
4. La aplicación espera al evento siguiente.

Nota Muchos eventos se producen junto con otros eventos. Por ejemplo, cuando se produce el evento DblClick, se producen también los eventos MouseDown, MouseUp y Click.

Eventos de los formularios

Como objetos que son, los formularios pueden ejecutar métodos y responder a eventos. Si observamos la lista de Procedimientos de la ventana de código correspondiente a un formulario, podemos observar los eventos que éste reconoce y a los cuales responde.



Los eventos que ocurren cuando se carga el formulario son:

Evento	Ocurre cuando
Initialize	Una aplicación crea un ejemplar de un formulario
Load	Se carga un formulario
Resize	El formulario se muestra o cambia el estado del mismo
Activate	El formulario pasa a ser la ventana activa. se produce siempre que un formulario se convierte en el formulario activo; el evento **Deactivate** se produce cuando otro formulario u otra aplicación se convierte en activo. Estos eventos son adecuados para iniciar o finalizar acciones del formulario. *Por ejemplo, en el evento Activate podría escribir código para resaltar el texto de un determinado cuadro de texto; con el evento Deactivate podría guardar los cambios efectuados en un archivo o en una base de datos.*
Paint	Un formulario se expone total o parcialmente después de haberse movido o ampliado, o después de haberse movido otro formulario que lo estaba cubriendo.

Eventos que ocurren cuando se cierra el formulario:

Evento	Ocurre cuando
QueryUnload	Se cierra un formulario, pero antes de producirse este hecho.
Unload	El formulario está a punto de cerrarse.
Terminate	Todas las variables que hacen referencia al formulario pasan a valer **Nothing** o cuando la última referencia al formulario queda fuera de su ámbito.

Eventos que pueden ocurrir en un formulario o en un control

Evento	Ocurre cuando
KeyDown	El usuario pulsa una tecla
KeyUp	El usuario suelta la tecla pulsada
KeyPress	El usuario pulsa una tecla; este evento ocurre después del evento KeyDown y antes de KeyUp
MouseDown	El usuario pulsa un botón del ratón

MouseUp	El usuario suelta el botón pulsado del ratón
Click	El usuario pulsa y suelta un botón del ratón

Hacer clic en los botones para realizar acciones

La forma más sencilla de permitir al usuario interactuar con una aplicación es proporcionarle un botón (CommandButton) para que haga clic en él. Puede usar el control de botón de comando que proporciona Visual Basic o crear su propio "botón" mediante un control de imagen que contenga un gráfico, por ejemplo un icono.

Cuando el usuario hace clic en el botón, éste no solamente realiza una acción, sino que, además, parece que se está presionando y soltando. Siempre que el usuario hace clic en un botón se invoca el procedimiento de evento **Click**. *(Para realizar cualquier acción que desee puede escribir código en el procedimiento de evento Click.)*

Hay muchas formas de elegir un botón de comando en tiempo de ejecución:

- Usar un *mouse* (ratón) para hacer clic en el botón.
- Mover el enfoque al botón presionando la tecla TAB y elegir luego el botón presionando la BARRA ESPACIADORA o ENTRAR.
- Presionar una tecla de acceso a un botón de comando (ALT+ la letra subrayada).
- Asignar a la propiedad **Value** del botón de comando el valor **True** en el código:

```
cmdCerrar.Value = True
```

- Invocar el evento Click del botón de comando en el código:

```
cmdCerrar_Click
```

Todas estas acciones hacen que Visual Basic invoque el procedimiento de evento Click.

Descripción del enfoque

El *enfoque* es la capacidad de recibir datos del usuario a través del *mouse* o del teclado. Cuando un objeto tiene el enfoque, puede recibir datos del usuario. En la interfaz de Microsoft Windows puede haber en ejecución varias aplicaciones a la vez, pero sólo la aplicación que tiene el enfoque tendrá una barra de título activa y podrá recibir datos del usuario. En un formulario de Visual Basic con varios cuadros de texto, sólo el cuadro de texto que tiene el enfoque mostrará el texto escrito desde el teclado.

Los **eventos *GotFocus*** y ***LostFocus*** se producen cuando un objeto recibe o pierde el enfoque. Los formularios y la mayoría de los controles admiten estos eventos.

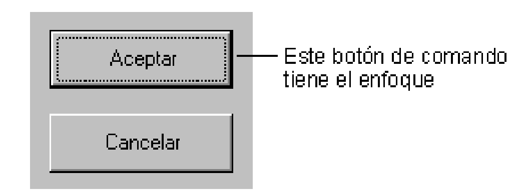
Evento	**Descripción**
GotFocus	Se produce cuando un objeto recibe el enfoque.
LostFocus	Se produce cuando un objeto pierde el enfoque. El procedimiento de evento LostFocus se usa principalmente para actualizar la comprobación y validación, o para invertir o modificar las condiciones establecidas en el procedimiento GotFocus del objeto.

Hay varias formas de dar el enfoque a un objeto:

- Seleccionar el objeto en tiempo de ejecución.

- Usar una tecla de acceso directo para seleccionar el objeto en tiempo de ejecución.
- Usar el método **SetFocus** en el código.

Puede ver cuándo algunos objetos tienen el enfoque. Por ejemplo, cuando los botones de comando tienen el enfoque, aparecen con un borde resaltado alrededor del título.



Un objeto sólo puede recibir el enfoque si sus propiedades **Enabled** y **Visible** tienen el valor **True**.

> **Nota** Un formulario sólo puede recibir el enfoque si no contiene controles que puedan recibir el enfoque.

Controles ligeros que no pueden recibir el enfoque son:

- Control Frame
- Control Image
- Control Label
- Control Line
- Control Shape

Configurar el orden de tabulación

El *orden de tabulación* es el orden en que un usuario se mueve de un control a otro mediante la tecla TAB. Cada formulario tiene su propio orden de tabulación. Normalmente, el orden de tabulación es igual al orden en que se han creado los controles.

Para cambiar el orden de tabulación de un control, asigne a la propiedad **TabIndex**.

La propiedad **TabIndex** de un control determina el lugar que ocupa en el orden de tabulación. De forma predeterminada, el primer control dibujado tiene un valor **TabIndex** de 0, el segundo un valor **TabIndex** de 1, etc. Cuando se modifica la posición de un control en el orden de tabulación, Visual Basic vuelve a numerar automáticamente las posiciones en el orden de tabulación de los demás controles para reflejar lo que se ha insertado o eliminado.

> El valor más alto de **TabIndex** es siempre una unidad menos que el número de controles que hay en el orden de tabulación (porque la numeración comienza en 0). Incluso aunque asigne a la propiedad **TabIndex** un número mayor que el número de controles, Visual Basic vuelve a convertir el valor al número de controles menos 1.
>
> **Nota** Los controles que no pueden obtener el enfoque, al igual que los controles desactivados o invisibles, no tienen una propiedad **TabIndex** y no están incluidos en el orden de tabulación. Cuando el usuario presiona la tecla TAB, se saltan estos controles

Quitar un control del orden de tabulación

Puede quitar un control del orden de tabulación si asigna a su propiedad **TabStop** el valor **False** (0). Un control cuya propiedad **TabStop** tiene el valor **False** sigue manteniendo su posición en el orden de tabulación real, incluso aunque se salte el control al ir de un control a otro con la tecla TAB.

Nota Un grupo de botones de opción tiene una única tabulación. El botón seleccionado (es decir, el botón cuya propiedad **Value** es **True**) tiene la propiedad **TabStop** establecida

automáticamente como **True**, mientras que los demás botones tienen la propiedad **TabStop** como **False**.

Validar datos de control restringiendo el enfoque

Los controles tienen también un ***evento*** **Validate**, que ocurre antes de que un control pierda el enfoque. Sin embargo, este evento ocurre sólo cuando la ***propiedad*** **CausesValidation** del control que va a recibir el enfoque tiene el valor True. En muchos casos, como el evento Validate sucede antes de que se pierda el enfoque, es más adecuado que el evento LostFocus para validar los datos, ya que mediante el evento Validate se puede impedir el cambio de enfoque a otro control hasta que se hayan cumplido todas las reglas de validación.

El evento Validate y la propiedad **CausesValidation** se usan conjuntamente para comprobar la entrada a un control antes de permitir que el usuario mueva el enfoque fuera de ese control. Por ejemplo, imagine una aplicación con varios cuadros de texto y un botón de Ayuda. Cuando cada uno de los cuadros de texto recibe el enfoque, quiere impedir que el usuario cambie dicho enfoque hasta que se cumplan los criterios de validación de ese cuadro de texto en particular; sin embargo, también quiere que los usuarios puedan pulsar el botón de Ayuda en cualquier momento. Para hacer esto, establezca el criterio de validación del evento Validate y la propiedad **CausesValidation** del botón **Ayuda** a **False**. Si la propiedad está establecida a **True** (el valor predeterminado), el evento Validate ocurrirá en el primer control. Si la propiedad está establecida a **False**, se impedirá la entrada de antemano del evento Validate del primer control.

Usos posibles

- Una aplicación para entrada de datos necesita llevar a cabo una validación de entrada de datos más sofisticada que la que pueda proporcionar el control **Masked Edit**, o la validación se produce en una norma del negocio.
- Un formulario debe impedir que los usuarios salgan de un control mediante la tecla TAB o una tecla aceleradora hasta que se hayan introducido datos en un campo.
- Un documento ActiveX que se ejecute en Internet Explorer necesita un método para que el usuario finalice una operación en el formulario antes de que la secuencia cambie el enfoque mediante programación.

Control del enfoque en el evento Validate

El evento Validate incluye un argumento *keepfocus (de mantenimiento del enfoque)*. Cuando el argumento esté establecido a **True**, el control retendrá el enfoque. Esto impide de manera eficaz que el usuario haga clic sobre cualquier otro control.

Mecánica de la escritura de código

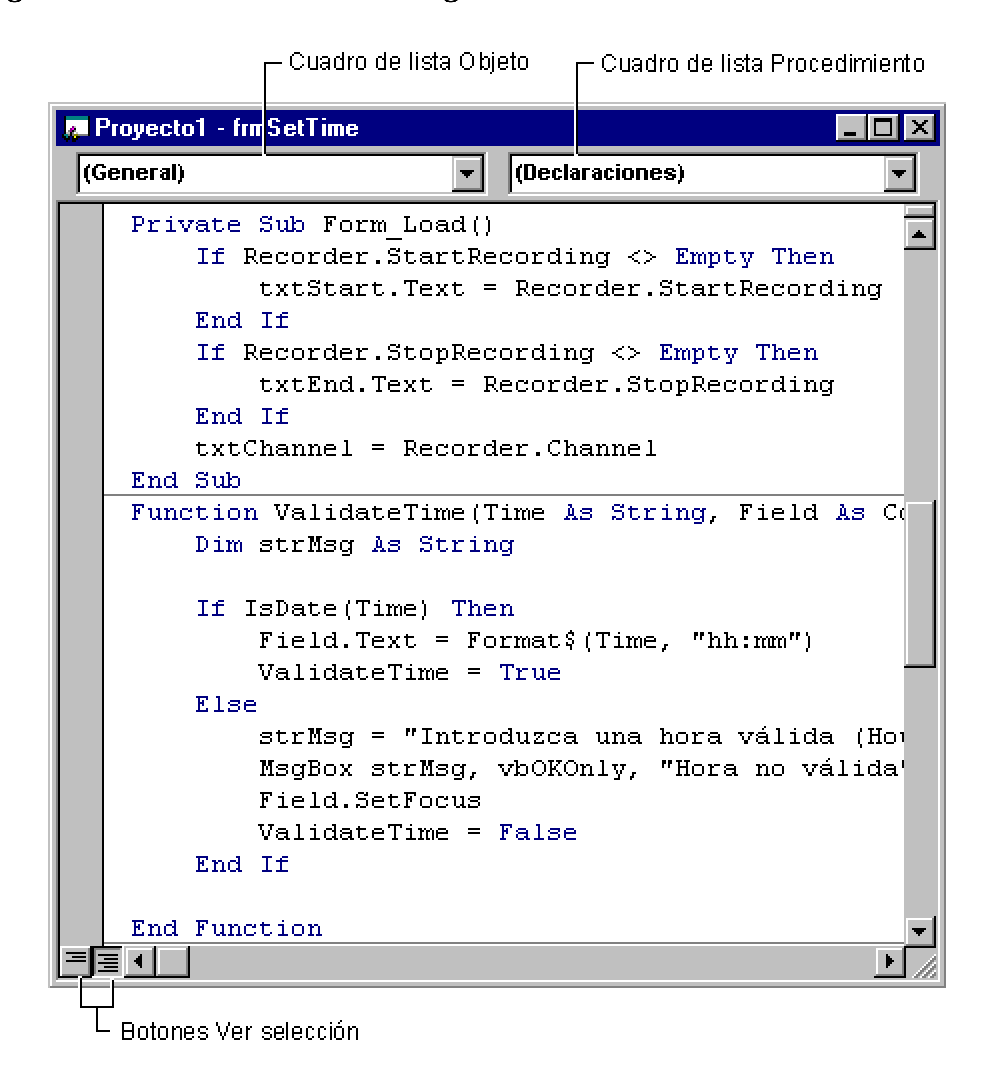
Las aplicaciones sencillas pueden consistir en un único formulario y todo el código de la aplicación reside en ese módulo de formulario. A medida que sus aplicaciones vayan creciendo y siendo más sofisticadas, agregará formularios adicionales. A veces tendrá código común que deseará ejecutar en varios formularios. No querrá duplicar el código en ambos formularios, por lo que creará un módulo independiente que contenga un procedimiento que ejecuta el código común. Este módulo

independiente debe ser un módulo estándar. Con el tiempo, puede construir una biblioteca de módulos que contenga los procedimientos compartidos.

Usar el Editor de código

El Editor de código de Visual Basic es una ventana donde se escribe la mayor parte del código. Es como un procesador de textos muy especializado con cierto número de características que facilita en gran medida la escritura de código en Visual Basic.

Fig. La ventana del Editor de código



Como trabaja con el código de Visual Basic en módulos, se abre una ventana distinta del Editor de código por cada módulo que seleccione en el Explorador de proyectos. El código de cada módulo está subdividido en distintas secciones para cada objeto del módulo. Para pasar de una sección a otra se utiliza el cuadro de lista **Objeto**. En un módulo de formulario, la lista incluye una sección general, una sección para el formulario propiamente dicho y una sección para cada control del formulario. Para un módulo de clase, la lista incluye una sección general y una sección de clase; para un módulo estándar sólo se muestra una sección general.

Cada sección de código puede contener varios procedimientos distintos, a los que se tiene acceso mediante el cuadro de lista **Procedimiento**. La lista de procedimientos de un módulo de formulario contiene una sección distinta para cada procedimiento de evento del formulario o el control. Por ejemplo, la lista de procedimientos de un control **Label** incluye secciones para los eventos Change, Click y DblClick, entre otros. Los módulos de clase sólo muestran los procedimientos de evento para la propia clase (Initialize y Terminate). Los módulos estándar no muestran ningún procedimiento de evento, ya que un módulo estándar no acepta eventos.

La lista de procedimientos de la sección general de un módulo contiene una única selección, la sección de declaraciones, donde se colocan las declaraciones a nivel de módulo de variables, constantes y DLL. Según vaya agregando procedimientos **Sub** o **Function** a un módulo, dichos procedimientos se agregan al cuadro de lista **Procedimiento** bajo la sección de declaraciones.

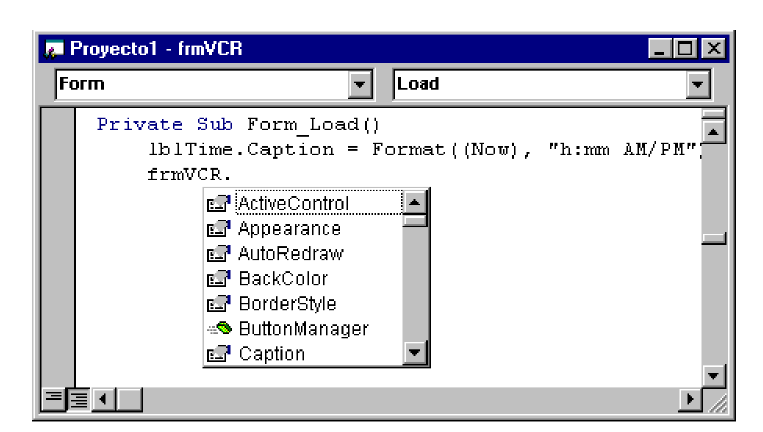
En la ventana del Editor de código hay disponibles dos vistas distintas del código. Puede elegir ver un único procedimiento cada vez o ver todos los procedimientos del módulo, estando cada procedimiento separado del siguiente mediante una línea. Para cambiar entre las dos vistas, utilice los botones **Ver selección** de la esquina inferior izquierda de la ventana del editor.

Terminar el código de forma automática

Visual Basic hace que la escritura de código sea mucho más sencilla mediante características que llenan automáticamente instrucciones, propiedades y argumentos. A medida que va introduciendo código, el editor muestra listas de elecciones adecuadas, prototipos de instrucciones o funciones, o valores. Hay opciones para activar o desactivar estas y otras configuraciones de código en la ficha **Editor** del cuadro de diálogo **Opciones**, al que se tiene acceso a través del comando **Opciones** del menú **Herramientas**.

Cuando especifica el nombre de un control en el código, la característica Lista de miembros automática presenta una lista desplegable con las propiedades disponibles para ese control. Escriba las primeras letras del nombre de la propiedad y se seleccionará en la lista; la tecla TAB completará la escritura. Esta opción resulta también muy útil cuando no sepa con seguridad qué propiedades hay disponibles para un determinado control. Incluso aunque haya elegido desactivar la característica Lista de miembros automática, podrá tener acceso a esta característica mediante la combinación de teclas Ctrl-J.

Fig. La característica Lista de miembros automática



La característica Información rápida automática muestra la sintaxis de las instrucciones y funciones. Cuando escriba el nombre de una instrucción o función válida de Visual Basic se mostrará la sintaxis inmediatamente debajo de la línea actual, con el primer argumento en negrita. Una vez que haya escrito el valor del primer argumento, se mostrará en negrita el segundo argumento. También se puede tener acceso a Información rápida automática con la combinación de teclas Ctrl-I.

Fig. Información rápida automática

Proyecto1 - frmSetTime

(General) | ValidateTime

```
Function ValidateTime(Time As String, Field As Control) As B
    Dim strMsg As String

    If IsDate(Time) Then
        Field.Text = Format$(Time, "h:mm AM/PM")
        ValidateTime = True
    Else
        strMsg = "¡Escriba una hora válida (Hora: Minutos AM
        MsgBox
        MsgBox(Prompt, [Buttons As VbMsgBoxStyle], [Title], [HelpFile], [Context])
        As VbMsgBoxResult
```

Marcadores

Puede usar marcadores para marcar líneas de código en el editor de código para que pueda volver más tarde a ellas. Lo comandos para alternar marcadores y para desplazarse por los marcadores existentes están disponibles en el menú Edición, elemento del menú Marcadores, o en la barra de herramientas Edición.

Conceptos básicos del código

Esta sección presenta información acerca de la mecánica de escritura de código, incluyendo dividir y combinar líneas de código, agregar comentarios al código, usar números en el código y seguir las convenciones de nomenclatura en Visual Basic.

Dividir una única instrucción en varias líneas

Puede dividir una instrucción larga en varias líneas en la ventana del Editor de código si utiliza el *carácter de continuación de línea* (un espacio en blanco seguido de un signo de subrayado). La utilización de este carácter puede hacer que sea más fácil leer el código, tanto en la pantalla como impreso en papel. El código siguiente se ha dividido en tres líneas mediante caracteres de continuación de línea (_):

```
Data1.RecordSource = _
"SELECT * FROM Titles, Publishers" _
& "WHERE Publishers.PubId = Titles.PubID _
& "AND Publishers.State = 'CA'"
```

No puede poner un comentario después de un carácter de continuación de línea en la misma línea. También hay algunas limitaciones sobre dónde se puede usar el carácter de continuación de línea.

Combinar instrucciones en una línea

Normalmente hay una instrucción de Visual Basic por línea y no hay ningún terminador de instrucción. Sin embargo, puede colocar dos o más instrucciones en una línea si utiliza un signo de dos puntos (:) para separarlas:

```
Text1.Text = "Hola" : Red = 255 : Text1.BackColor = _
Red
```

Sin embargo, para hacer el código más legible, es mejor colocar cada instrucción en una línea distinta.

Agregar comentarios al código

El símbolo de comentario (') indica a Visual Basic que pase por alto las palabras que van a continuación de él. Estas palabras son comentarios situados en el código para el desarrollador y otros programadores que vayan a examinar después el código. Por ejemplo:

```
' Este comentario comienza en el borde izquierdo de
' la pantalla.
Text1.Text = "Hola"          ' Pone un saludo amistoso
                            ' en el cuadro de texto.
```

Los comentarios pueden seguir a una instrucción en la misma línea o pueden ocupar una línea completa. Ambos se ilustran en el código anterior. **Recuerde que los comentarios no pueden ir detrás de un carácter de continuación de línea en la misma línea.**

Descripción de los sistemas de numeración

La mayoría de los números de este documento son decimales (base 10). Pero en ocasiones es conveniente usar números hexadecimales (base 16) o números octales (base 8). Visual Basic representa los números hexadecimales con el prefijo &H y los octales con &O. La tabla siguiente muestra los mismos números en formato decimal, octal y hexadecimal.

Decimal	Octal	Hexadecimal
9	&O11	&H9
15	&O17	&HF
16	&O20	&H10
20	&O24	&H14
255	&O377	&HFF

Normalmente no tendrá que aprender los sistemas hexadecimal u octal ya que el equipo puede trabajar con números en cualquier sistema. Sin embargo, algunos sistemas de numeración se prestan a determinadas tareas, como la utilización de números hexadecimales para configurar los colores de la pantalla y los controles.

Introducción a los procedimientos

Recordar que el código en Visual Basic se almacena en módulos (de formulario, estándar y de clase). Cada módulo puede contener declaraciones y procedimientos.

Con los procedimientos puede simplificar las tareas de programación, ya que divide los programas en componentes lógicos más pequeños que pueden convertirse en bloques básicos que le permiten mejorar y ampliar Visual Basic.

Los procedimientos resultan muy útiles para condensar las tareas repetitivas o compartidas, como cálculos utilizados frecuentemente, manipulación de texto y controles, y operaciones con bases de datos. La base de una aplicación en Visual Basic la forman sus procedimientos conducidos por eventos.

Hay dos ventajas principales cuando se programa con procedimientos:

- Los procedimientos le permiten dividir los programas en unidades lógicas discretas, cada una de las cuales se puede depurar más fácilmente que un programa entero sin procedimientos.
- Los procedimientos que se utilizan en un programa pueden actuar como bloques de construcción de otros programas, normalmente con pocas o ninguna modificación.

En Visual Basic se utilizan varios tipos de procedimientos, dos de ellos son:

- Procedimientos **Sub** que no devuelven un valor.
- Procedimientos **Function** que devuelven un valor.

Un procedimiento **Sub**, o **Function** contiene partes de código que se pueden ejecutar como una unidad.

Procedimientos Sub

Un procedimiento **Sub** es un bloque de código que se ejecuta como respuesta a un evento. Al dividir el código de un módulo en procedimientos **Sub**, es más sencillo encontrar o modificar el código de la aplicación.

La sintaxis de un procedimiento **Sub** es la siguiente:

```
[Private|Public][Static]Sub nombreProcedimiento(parámetros)
        Instrucciones
        [Exit Sub]
        Instrucciones
End Sub
```

Cada vez que se llama al procedimiento se ejecutan las *instrucciones* que hay entre **Sub** y **End Sub**. Se pueden colocar los procedimientos **Sub** en módulos estándar y módulos de formulario. De forma predeterminada, los procedimientos **Sub** son **Public** en todos los módulos, lo que significa que se les puede llamar desde cualquier parte de la aplicación.

Los *parámetros* de un procedimiento son como las declaraciones de variables; se declaran valores que se pasan desde el procedimiento que hace la llamada. *(Ver implicaciones del paso de parámetros)*

Para hacer que todas las variables locales de un procedimiento (Sub o Function), sean por omisión estáticas, hay que colocar al principio de la cabecera del procedimiento la palabra clave **Static**. Por ejemplo:

```
Public Static Sub Proc_1(X AS Double, N AS Integer)
  ...
End Sub
```

Para hacer que un procedimiento (Sub o Function) sólo sea accesible desde los procedimientos del módulo al cual pertenece, hay que colocar al principio de la cabecera del procedimiento la palabra clave **Private**. Ejemplo:

```
Private Sub Proc_1(X AS Double, N AS Integer)
  ...
End Sub
```

Si no se especifica la palabra clave **Private** se supone que el procedimiento es **Public**, lo que significa que puede ser invocado desde otros módulos.

Resulta muy útil en Visual Basic distinguir entre dos tipos de procedimientos **Sub**, *procedimientos generales* y *procedimientos de evento*.

Salir de un procedimiento Sub

También puede salir de un procedimiento desde una estructura de control. **Exit Sub** puede aparecer tantas veces como sea necesario, en cualquier parte del cuerpo de un procedimiento **Sub**.

Exit Sub es muy útil cuando el procedimiento ha realizado todo lo que tenía que hacer y se quiere volver inmediatamente.

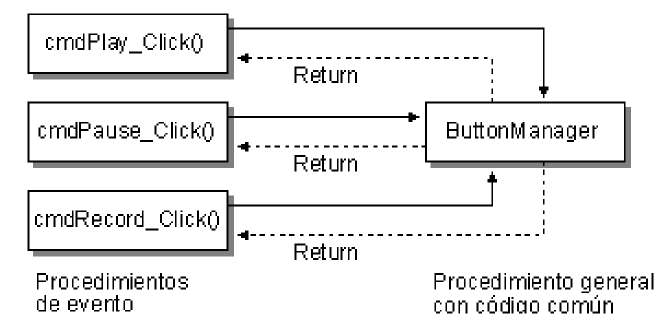
Procedimientos generales

Un procedimiento general indica a la aplicación cómo realizar una tarea específica. Una vez que se define un procedimiento general, se le debe llamar específicamente desde la aplicación.

Los procedimientos generales se crean porque, muchos procedimientos de evento distintos pueden necesitar que se lleven a cabo las mismas acciones. Es una buena estrategia de programación colocar las instrucciones comunes en un procedimiento distinto (un procedimiento general) y hacer que los procedimientos de evento lo llamen. Esto elimina la necesidad de duplicar código y también hace que la aplicación sea más fácil de mantener.

Ejemplo en la figura se muestra la utilización de un procedimiento general. El código de los eventos Click llama al procedimiento **Sub** ButtonManager que ejecuta su propio código y devuelve el control al procedimiento de evento Click.

Fig. Cómo llaman los procedimientos de evento a procedimientos generales



Ejemplo:

```
Public Sub Factorial(N AS Integer, F AS Long)
 If N=0 Then
      F=1
 Else
```

```
        F=1
        Do While N>0
              F=N*F
              N=N-1
        Loop
  End If
End Sub
```

Procedimientos de evento

Un procedimiento de evento permanece inactivo hasta que se le llama para responder a eventos provocados por el usuario o desencadenados por el sistema. Cuando un objeto en Visual Basic reconoce que se ha producido un evento, llama automáticamente al procedimiento de evento utilizando el nombre correspondiente al evento. Como el nombre establece una asociación entre el objeto y el código, se dice que los procedimientos de evento están adjuntos a formularios y controles.

- Un procedimiento de evento de un control combina el nombre real del control (especificado en la propiedad **Name**), un carácter de subrayado (_) y el nombre del evento. Por ejemplo, si desea que un botón de comando llamado *cmdPlay* llame a un procedimiento de evento cuando se haga clic en él, utilice el procedimiento *cmdPlay_Click*.
- Un procedimiento de evento de un formulario combina la palabra "*Form*", un carácter de subrayado y el nombre del evento. Si desea que un formulario llame a un procedimiento de evento cuando se hace clic en él, utilice el procedimiento *Form_Click*.. (Como los controles, los formularios tienen nombres únicos, pero no se utilizan en los nombres de los procedimientos de evento.) Si está utilizando el formulario MDI, el procedimiento de evento combina la palabra "*MDIForm*", un carácter de subrayado y el nombre del evento, como *MDIForm_Load*.

Todos los procedimientos de evento utilizan la misma sintaxis general:

Sintaxis de un evento de control	**Sintaxis de un evento de formulario**
Private Sub *nombrecontrolNombreevento* (*argumentos*) *instrucciones* End Sub	Private Sub Form_*nombreevento* (*argumentos*) *Instrucciones* End Sub

Aunque puede escribir procedimientos de evento nuevos, es más sencillo usar los procedimientos de código que facilita Visual Basic, que incluyen automáticamente los nombres correctos de procedimiento. Puede seleccionar una plantilla en la ventana Editor de código si selecciona un objeto en el cuadro **Objeto** y selecciona un procedimiento en el cuadro **Procedimiento**.

También es conveniente establecer la propiedad **Name** de los controles antes de empezar a escribir los procedimientos de evento para los mismos. Si cambia el nombre de un control tras vincularle un procedimiento, deberá cambiar también el nombre del procedimiento para que coincida con el nuevo nombre del control. De lo contrario, Visual Basic no será capaz de hacer coincidir el control con el procedimiento. Cuando el nombre de un

procedimiento no coincide con el nombre de un control, se convierte en un procedimiento general.

Trabajar con procedimientos

Para crear un procedimiento general nuevo

Escriba el encabezado de un procedimiento en la ventana Código y presione ENTRAR. El encabezado del procedimiento puede ser tan simple como **Sub** o **Function** seguido de un nombre. Por ejemplo, puede especificar cualquiera de los siguientes:

```
Sub ActualizarForm ()
Function ObtenerCoord ()
```

Visual Basic responde completando la plantilla del nuevo procedimiento.

Seleccionar procedimientos existentes

Para ver un procedimiento en el módulo actual:

Para ver un procedimiento general existente, seleccione **"(General)"** en el cuadro **Objeto** de la ventana Código y seleccione el procedimiento en el cuadro **Procedimiento**.

–o bien–

Para ver un procedimiento de evento, seleccione el objeto apropiado en el cuadro **Objeto** de la ventana Código y seleccione el evento en el cuadro **Procedimiento**.

Para ver un procedimiento de otro módulo

1. En el menú **Ver**, elija **Examinador de objetos**.
2. Seleccione el proyecto en el cuadro **Proyecto/Biblioteca**.
3. Seleccione el módulo en la lista **Clases** y el procedimiento en la lista **Miembros de**.
4. Elija **Ver definición**.

Llamar a procedimientos Sub

Un procedimiento **Sub** difiere de un procedimiento **Function** en que al procedimiento **Sub** no se le puede llamar mediante su nombre en una expresión. La llamada a un procedimiento **Sub** es una instrucción única. Además, un procedimiento **Sub** no devuelve un valor en su nombre como hace una función. Sin embargo, al igual que **Function**, un procedimiento **Sub** puede modificar los valores de las variables que se le pasan.

Hay dos formas de llamar a un procedimiento **Sub**:

```
' Ambas instrucciones llaman a un Sub denominado MiProc.
Call MiProc (PrimerArgumento, SegundoArgumento)
MiProc PrimerArgumento, SegundoArgumento
```

Observe que cuando utiliza la sintaxis **Call**, debe poner los argumentos entre paréntesis. Si omite la palabra clave **Call**, deberá también omitir los paréntesis alrededor de los argumentos.

Llamar a procedimientos en otros módulos

Se puede llamar a los procedimientos públicos de otros módulos desde cualquier parte del proyecto. Necesitará especificar el módulo que contiene el procedimiento al que está llamando. Las técnicas

para hacerlo varían, dependiendo de si el procedimiento está ubicado en un módulo estándar o de formulario.

Procedimientos en formularios

Todas las llamadas que se hacen desde fuera del módulo de formulario deben señalar al módulo de formulario que contiene el procedimiento. Si un procedimiento llamado *SomeSub* está en un módulo de formulario llamado *Form1*, puede llamar al procedimiento en Form1 mediante esta instrucción:

```
Call Form1.SomeSub(argumentos)
```

Procedimientos en módulos estándar

Si el nombre de un procedimiento es único, no necesita incluir el nombre del módulo en la llamada. Una llamada desde dentro o fuera del módulo hará referencia a ese procedimiento único. Un procedimiento es único si sólo aparece en un lugar.

Si dos o más módulos contienen un procedimiento con el mismo nombre, debe calificarlo con el nombre del módulo. Una llamada a un procedimiento común desde el mismo módulo ejecuta el procedimiento de ese módulo. Por ejemplo, con un procedimiento llamado *CommonName* en *Module1* y *Module2*, una llamada a *CommonName* desde *Module2* ejecutará el procedimiento *CommonName* de *Module2* y no el procedimiento *CommonName* de *Module1*.

En una llamada a un nombre de procedimiento común desde otro módulo, debe especificar el módulo del procedimiento. Por ejemplo, si desea llamar al procedimiento *CommonName* de *Module2* desde *Module1*, utilice:

```
Module2.CommonName(argumentos)
```

Introducción a las variables, constantes y tipos de datos

A menudo necesitará almacenar valores temporalmente cuando esté realizando cálculos con Visual Basic. Por ejemplo, puede que desee calcular diversos valores, compararlos y realizar distintas operaciones con ellos, dependiendo de los resultados de la comparación. Necesitará conservar los valores si desea compararlos, pero no necesitará almacenarlos en una propiedad.

Visual Basic, como la mayoría de los lenguajes de programación, utiliza *variables* para almacenar valores. Las variables tienen un nombre (la palabra que utiliza para referirse al valor que contiene la variable) y un *tipo de dato* (que determina la clase de datos que puede almacenar la variable). Se pueden usar *matrices* para almacenar colecciones indexadas de variables relacionadas.

Las constantes también almacenan valores pero, como su nombre indica, estos valores permanecen constantes durante la ejecución de la aplicación. La utilización de constantes puede hacer más legible el código ya que proporciona nombres significativos en vez de números. Visual Basic incorpora una serie de constantes, pero puede crear sus propias constantes.

Los tipos de datos controlan el almacenamiento interno de datos en Visual Basic.

Tipos de datos intrínsecos

Todas las variables tienen un tipo de dato que determina la clase de datos que pueden almacenar. El tipo de dato de la variable determina cómo se almacenan los bits que representan esos valores en la memoria del equipo.

Tipo de datos	Descripción	Tamaño de almacenamiento	Intervalo
Byte	Carácter	1 byte	0 a 255
Boolean	Booleano	2 bytes	**True** o **False**
Integer	Entero	2 bytes	-32,768 a 32,767
Long	Entero largo	4 bytes	-2,147,483,648 a 2,147,483,647
Single	Coma flotante/ precisión simple	4 bytes	-3,402823E38 a –1,401298E-45 para valores negativos; 1,401298E-45 a 3,402823E38 para valores positivos
Double	Coma flotante/ precisión doble	8 bytes	-1,79769313486232E308 a -4,94065645841247E-324 para valores negativos; 4,94065645841247E-324 a 1,79769313486232E308 para valores positivos
Currency	Entero a escala. Numero con punto decimal fijo	8 bytes	-922.337.203.685.477,5808 a 922.337.203.685.477,5807
Decimal	Números con 0 a 28 decimales (No se puede declarar una variable de este tipo. Solo se puede utilizar con un Variant)	14 bytes	+/-79.228.162.514.264.337.593.543.950.335 sin punto decimal; +/-7,9228162514264337593543950335 con 28 posiciones a la derecha del signo decimal; el número más pequeño distinto de cero es +/-0,0000000000000000000000000001
Date	Fecha/Hora	8 bytes	1 de enero de 100 a 31 de diciembre de 9999
Object	Referencia a un objeto	4 bytes	Cualquier referencia a tipo **Object**

Tipo de datos	Descripción	Tamaño de almacenamiento	Intervalo
String	Cadena de caracteres de longitud variable.	10 bytes + 1 byte por caracter	Desde 0 a 2.000 millones (Hasta 2^{32} caracteres aprox)
String	Cadena de caracteres de longitud fija.	1 byte por caracter	Desde 1 a 65.400 aproximadamente (hasta 64K aproximadamente)
Variant (por omisión)	Con números.	16 bytes	Cualquier valor numérico hasta el intervalo de un tipo **Double**
Variant (por omisión)	Con caracteres.	22 bytes + longitud de la cadena	El mismo intervalo que para un tipo **String** de longitud variable
Definido por el usuario (utilizando **Type**)		Número requerido por los elementos	El intervalo de cada elemento es el mismo que el intervalo de su tipo de datos.

Visual Basic proporciona varios tipos de datos numéricos: **Integer**, **Long** (entero largo), **Single** (signo flotante de simple precisión), **Double** (signo flotante de doble precisión) y **Currency**. Usar un tipo de dato numérico emplea normalmente menos espacio de almacenamiento que un tipo **Variant**.

Si sabe que una variable siempre va a almacenar números *enteros* (como 12) en vez de números fraccionarios (como 3,57), declárela como un tipo **Integer** o **Long**. Las operaciones con enteros son más rápidas y estos tipos consumen menos memoria que otros tipos de datos. Resultan especialmente útiles como variables de contador en bucles **For...Next**.

Si la variable contiene una *fracción*, declárela como variable **Single**, **Double** o **Currency**. El tipo de dato **Currency** acepta hasta cuatro dígitos a la derecha del separador decimal y hasta quince dígitos a la izquierda; es un tipo de dato de signo fijo adecuado para cálculos monetarios. Los números de signo flotante (**Single** y **Double**) tienen más intervalo que **Currency**, pero pueden estar sujetos a pequeños errores de redondeo.

Nota Los valores de signo flotante se pueden expresar como *mmm*__E__*eee* o *mmm*__D__*eee*, donde *mmm* es la mantisa y *eee* el exponente (potencia de 10). El valor positivo más alto de un tipo de dato **Single** es 3,402823E+38 ó 3,4 veces 10 a la 38[a] potencia; el valor positivo más alto de un tipo de dato **Double** es 1,79769313486232D+308 o alrededor de 1,8 veces 10 a la 308[a] potencia. Si utiliza **D** para separar la mantisa y el exponente en un literal numérico, el valor se tratará como un tipo de dato **Double**. Igualmente, usar **E** de la misma manera hace que el valor se trate como un tipo de dato **Single**.

Si la variable contiene datos binarios, declárela como matriz de tipo **Byte**. Usar variables **Byte** para almacenar datos binarios los preserva durante las conversiones de formato. Cuando se convierten las variables **String** entre los formatos ANSI y Unicode, los datos binarios de la variable resultan dañados. Todos los operadores que funcionan con enteros funcionan con el tipo de dato **Byte** excepto el de resta unaria. Puesto que **Byte** es un tipo sin signo con el intervalo 0-255, no puede representar un valor negativo. Así, para la resta unaria, Visual Basic convierte antes el tipo **Byte** en un entero con signo.

Si tiene una variable que siempre contendrá una cadena y nunca un valor numérico, puede declararla del tipo **String**.

Si tiene una variable que siempre contendrá solamente información del tipo verdadero y falso, sí y no o activado o desactivado, puede declararla del tipo **Boolean**. El valor predeterminado de **Boolean** es **False**.

Los valores de fecha y hora pueden almacenarse en el tipo de dato específico **Date** en variables **Variant**. En ambos tipos se aplican las mismas características generales a las fechas. Cuando se convierten otros tipos de datos numéricos en **Date**, los valores que hay a la izquierda del signo decimal representan la información de fecha, mientras que los valores que hay a la derecha del signo decimal representan la hora. Medianoche es 0 y mediodía es 0.5. Los números enteros negativos representan fechas anteriores al 30 de diciembre de 1899.

De forma predeterminada, si no proporciona un tipo de dato, la variable toma el tipo de dato **Variant**. El tipo de dato **Variant** es como un camaleón; puede representar diferentes tipos de datos en distintas situaciones, ya que capaz de almacenar todos los tipos de datos definidos en el sistema. No tiene que convertir estos tipos de datos cuando los asigne a una variable **Variant**: Visual Basic realiza automáticamente cualquier conversión necesaria. Es posible asignar todas las variables numéricas entre sí y a variables del tipo **Variant**. Visual Basic redondea en vez de truncar la parte fraccionaria de un número de signo flotante antes de asignarlo a un entero.

Sin embargo, si sabe que una variable almacenará siempre un tipo de dato determinado, ***Visual Basic tratará de forma más eficiente los datos si declara la variable con ese tipo.*** Los otros tipos de datos disponibles le permiten optimizar el código en cuanto a velocidad y tamaño cuando no necesite la flexibilidad que proporciona el tipo **Variant**.

Si bien puede realizar operaciones con variables **Variant** sin saber exactamente el tipo de dato que contienen, hay algunas trampas que debe evitar.

- Si realiza operaciones aritméticas o funciones sobre un **Variant**, el **Variant** debe contener un número.
- Si está concatenando cadenas, utilice el operador **&** en vez del operador **+**.

Además de poder actuar como otros tipos de datos estándar, los **Variant** también pueden contener tres valores especiales: **Empty**, **Null** y **Error**.

Los tipos de datos se aplican a otras cosas además de a las variables. Cuando asigna un valor a una propiedad, dicho valor tiene un tipo de dato; los argumentos de las funciones tienen también tipos de datos. De hecho, todo lo relacionado con datos en Visual Basic tiene un tipo de dato.

También puede declarar matrices de cualquiera de los tipos fundamentales.

Nota Las matrices de cualquier tipo de datos requieren 20 bytes de memoria más cuatro bytes para cada dimensión de matriz, más el número de bytes que ocupan los propios datos. Puede calcular la memoria que ocupan los datos multiplicando el número de elementos de datos por el tamaño de cada elemento. Por ejemplo, los datos de una matriz unidimensional que consten de cuatro elementos de datos tipo **Integer** de dos bytes cada uno, ocupan ocho bytes. Los ocho bytes que requieren los datos más los 24 bytes necesarios para la matriz suman un requisito total de memoria de 32 bytes para dicha matriz. Un tipo **Variant** que contiene una matriz requiere 12 bytes más que la matriz por sí sola.

Valores especiales que pueden contener las variables de tipo Variant

El valor Empty

A veces necesitará saber si se ha asignado un valor a una variable existente. Una variable **Variant** tiene el valor **Empty** antes de asignarle un valor. El valor **Empty** es un valor especial distinto de 0, una cadena de longitud cero ("") o el valor **Null**. Puede probar el valor **Empty** con la función **IsEmpty**:

```
If IsEmpty(Z) Then Z = 0
```

Cuando un **Variant** contiene el valor **Empty**, puede usarlo en expresiones; se trata como un 0 o una cadena de longitud cero, dependiendo de la expresión. El valor **Empty** desaparece tan pronto como se asigna cualquier valor (incluyendo 0, una cadena de longitud cero o **Null**) a una variable **Variant**. Puede establecer una variable **Variant** de nuevo como **Empty** si asigna la palabra clave **Empty** al **Variant**.

El valor Null

Null. **Null** se utiliza comúnmente en aplicaciones de bases de datos para indicar datos desconocidos o que faltan. Debido a la forma en que se utiliza en las bases de datos, **Null** tiene algunas características únicas:

- Las expresiones que utilizan **Null** dan como resultado siempre un **Null**. Así, se dice que **Null** se "propaga" a través de expresiones; si cualquier parte de la expresión da como resultado un **Null**, la expresión entera tiene el valor **Null**.
- Al pasar un **Null**, un **Variant** que contenga un **Null** o una expresión que dé como resultado un **Null** como argumento de la mayoría de las funciones hace que la función devuelva un **Null**.
- Los valores **Null** se propagan a través de funciones intrínsecas que devuelven tipos de datos **Variant**.

También puede asignar un **Null** mediante la palabra clave **Null**:

```
Z = Null
```

Puede usar la función **IsNull** para probar si una variable **Variant** contiene un **Null**:

```
If IsNull(X) And IsNull(Y) Then
   Z = Null
Else
   Z = 0
End If
```

Si asigna **Null** a una variable de un tipo que no sea **Variant**, se producirá un error interceptable. Asignar **Null** a una variable **Variant** no provoca ningún error y el **Null** se propagará a través de expresiones que contengan variables **Variant** (**Null** no se propaga a través de determinadas funciones). Puede devolver **Null** desde cualquier procedimiento **Function** con un valor de devolución de tipo **Variant**.

Null no se asigna a las variables a menos que se haga explícitamente, por lo que si no utiliza **Null** en su aplicación, no tendrá que escribir código que compruebe su existencia y lo trate.

El valor Error

En un **Variant**, **Error** es un valor especial que se utiliza para indicar que se ha producido una condición de error en un procedimiento. Sin embargo, a diferencia de otros tipos de error, no se produce el tratamiento de errores a nivel normal de aplicación. Esto le permite a

usted o a la propia aplicación elegir alternativas basadas en el valor del error. Los valores de error se crean convirtiendo números reales en valores de error mediante la función **CVErr**.

Declaración de variables

Puede considerar una variable como un marcador de posición en memoria de un valor desconocido. Declarar una variable es decirle al programa algo de antemano, por eso, antes de utilizar una variable es aconsejable declarar su tipo de dato.

Por ejemplo, suponga que está creando un programa para una frutería que haga un seguimiento del precio de las manzanas. No sabe el precio de una manzana o la cantidad que se ha vendido hasta que no se produce realmente la venta. Puede usar dos variables para almacenar los valores desconocidos (vamos a llamarlos PrecioManzanas y ManzanasVendidas). Cada vez que se ejecuta el programa, el usuario especifica los valores de las dos variables. Para calcular las ventas totales y mostrarlas en un cuadro de texto llamado txtVentas, el código debería parecerse al siguiente:

```
txtVentas.txt = PrecioManzanas * ManzanasVendidas
```

La expresión devuelve un total distinto cada vez, dependiendo de los valores que indique el usuario. Las variables le permiten realizar un cálculo sin tener que saber antes cuáles son los valores especificados.

En este ejemplo, el tipo de dato de PrecioManzanas es **Currency**; el tipo de dato de ManzanasVendidas es **Integer**. Las variables pueden representar otros muchos valores como valores de texto, fechas, diversos tipos numéricos e incluso objetos.

Una forma en que se declara una variable es mediante la instrucción **Dim**, proporcionando un nombre a la variable:

```
Dim nombreVariable [As tipo]
```

La cláusula opcional **As** *tipo* de la instrucción **Dim** le permite definir el tipo de dato o de objeto de la variable que va a declarar.

NOTA: En la sintaxis los corchetes indican información que es opcional.

Ejemplos de declaración:

```
Dim I AS Integer
Dim R As Double
Dim F As Currency
```

En una sentencia Dim puede hacer mas de una declaración

```
Dim L As Long, X As Currency
```

Cualquier declaración de este tipo inicia las variables numéricas con el valor cero y las variables alfanuméricas con el carácter nulo.

De forma predeterminada, una variable o argumento de cadena es una *cadena de longitud variable*; la cadena crece o disminuye según le asigne nuevos datos.

```
Dim Nombre AS String
```

También puede declarar cadenas de longitud fija. Especifique una *cadena de longitud fija* con esta sintaxis:

```
String * tamaño
```

Por ejemplo, para declarar una cadena que tiene siempre 50 caracteres de longitud, utilice un código como este:

```
Dim NombreEmp As String * 50
```

Si asigna una cadena con menos de 50 caracteres, `NombreEmp` se rellenará con espacios en blanco hasta el total de 50 caracteres. Si asigna una cadena demasiado larga a una cadena de longitud fija, Visual Basic simplemente truncará los caracteres. Puesto que las cadenas de longitud fija se rellenan con espacios al final, verá que las funciones **Trim** y **RTrim**, que quitan los espacios en blanco, resultan muy útiles cuando se trabaja con ellas.

Cuando se declara una variable y no se especifica su tipo, se asume que es de tipo Variant:

```
Dim A, B As Integer
```

En esta sentencia parecería que A y B son Integer pero no es asi, A es de tipo Variant (por omisión) y B es de tipo Integer.

Cuando una variable se utiliza y no se declara explícitamente, se asume que es de tipo **Variant**.

```
L="Dato:"    'variable de tipo String
L=3.254546   'variable de tipo Double
```

Suponiendo que L no fue declarada explícitamente, entonces lo anterior indica que L es una variable **Variant** que ha cambiado su tipo para comportarse como una cadena de caracteres, y a continuación vuelve a cambiar su tipo para comportarse como una variable real de precisión doble. Aunque no es necesario declarar una variable antes de utilizarla, esta forma de trabajar puede ser una fuente de errores. Para evitar errores, podemos indicarle a Visual Basic que genere un mensaje de error siempre que encuentre una variable no declarada explícitamente.

> En la ficha **Editor** del cuadro de diálogo **Opciones** (menú Opciones – Tools), active la opción **Declaración de variables requerida**. La instrucción **Option Explicit** requiere que declare todas las variables del programa de Visual Basic.

Alcance de las variables

Cuando declara una variable en un procedimiento, sólo el código de dicho procedimiento puede tener acceso o modificar el valor de la variable; tiene un alcance que es ***local*** al procedimiento. A veces, sin embargo, se necesita usar una variable con un alcance más general, como aquella cuyo valor está disponible para todos los procedimientos del mismo módulo o incluso para todos los procedimientos de toda la aplicación, se conocen como ***globales***. Visual Basic le permite especificar el alcance de una variable cuando la declara.

El ***alcance de una variable*** *define* qué partes del código son conscientes de su existencia, es decir, el espacio de la aplicación donde la variable es visible y por lo tanto se puede utilizar.

Ámbito	**Declaración**	**Visible en**
Local	**Dim**, **Static**, o **ReDim** (dentro de un procedimiento, subprocedimiento o función)	El procedimiento en el que está declarada.
Módulo	**Dim** o **Private** (sección de *declaraciones* del módulo de formulario o de código [.frm, .bas])	Todos los procedimientos del módulo de formulario o de código
Global	**Public** (sección de declaraciones del un módulo de código [.bas])	En toda la aplicación. No puede declarar variables públicas dentro de un procedimiento.

Las variables que se declaran en *un procedimiento* mediante la instrucción **Dim**:

- Sólo existen mientras se ejecuta el procedimiento;
- Su valor es local al procedimiento, es decir, no puede tener acceso a una variable de un procedimiento desde otro procedimiento.
- Cuando termina el procedimiento, desaparece el valor de la variable, es decir, no conserva su valor entre una llamada al procedimiento y la siguiente; es reiniciada cada vez que se entra en el procedimiento.

Estas características le permiten usar los mismos nombres de variables en distintos procedimientos sin preocuparse por posibles conflictos o modificaciones accidentales.

```
Private I As Integer
Dim Cantidad As Double
```

Las variables que se declaran en un procedimiento mediante la instrucción **Static** existen mientras se ejecuta la aplicación, es decir, no se pierde su valor cuando termina el procedimiento, solamente son iniciadas la primera vez, y la siguiente vez que es llamado esa variable conserva su valor.

```
Static num1 AS Integer
Static SuNombre As String
```

Por ejemplo, la siguiente función calcula un total continuo sumando un valor nuevo al total de los valores almacenados previamente en la variable estática `ManzanasVendidas`:

```
Function EjecuciónTotal(num)
   Static ManzanasVendidas
   ManzanasVendidas = ManzanasVendidas + num
   EjecuciónTotal = ManzanasVendidas
End Function
```

Si se hubiera declarado `ManzanasVendidas` con **Dim** en vez de con **Static**, no se habrían preservado los valores acumulados previamente en las llamadas a la función y la función devolvería simplemente el mismo valor con el que se la llama. Puede obtener el mismo resultado si declara `ManzanasVendidas` en la sección Declaraciones del módulo, haciendo que sea una variable a nivel de módulo. Sin embargo, una vez que cambie el alcance de una variable de esta forma, el procedimiento no tendrá acceso exclusivo a la variable. Puesto que otros procedimientos pueden tener acceso y modificar el valor de la variable, los totales continuos no serían fiables y sería más difícil mantener el código.

Para hacer que todas las variables de un procedimiento sean estáticas, puede también proceder declarando el procedimiento estático. Ejemplo:

```
Private Static Sub Form_Load()
 …
End Sub
```

Esto hace que todas las variables del procedimiento sean estáticas sin tener en cuenta si se declararon con **Static**, **Dim**, **Private** o de forma implícita. Puede colocar **Static** delante de cualquier encabezado de procedimientos **Sub** o **Function**, incluyendo los procedimientos de eventos y aquellos que se declararon como **Private**.

A nivel de *modulo* no hay diferencia entre **Dim y Private**, pero se aconseja utilizar prívate en contraste con **Public**. Este tipo de variables por omisión son estáticas. Una variable tiene alcance de *nivel de módulo* si se declara como **Private** en un módulo estándar o en un módulo de formulario, respectivamente.

Nota La coherencia es crucial para usar esta técnica de forma productiva; el corrector de sintaxis de Visual Basic no interceptará las variables de nivel de módulo que comiencen con "p".

```
Private I As Integer, Amt As Double
Private SuNombre As String, PagadoPorJuan As Currency
Private Prueba, Cantidad, J As Integer
```

Nota: Recuerde que si no proporciona un tipo de dato, se asigna a la variable el tipo predeterminado. En el ejemplo anterior, las variables `Prueba` y `Cantidad` tienen un tipo de dato **Variant**.

Una variable tiene alcance global si se declara como **Public** en un módulo estándar o en un módulo de formulario.

```
Public PagadoPorJuan As Currency
```

Pueden ser accedidas desde cualquier modulo. Para acceder a la variable, es necesario especificar su pertenencia; es decir, de qué objeto es dato miembro dicha variable. Por ejemplo si la variable *Conta* se declaro como globla en el formularo *Form1*:

```
Form1.Conta
```

Las variables se deben definir siempre con el menor alcance posible. Las **variables globales** (públicas) pueden crear máquinas de estado enormemente complejas y hacer la lógica de una aplicación muy difícil de entender. Así mismo, las variables globales también hacen mucho más difícil mantener y volver a usar el código. En una aplicación de Visual Basic, las variables globales se deben usar sólo cuando no exista ninguna otra forma cómoda de compartir datos entre formularios. Cuando haya que usar variables globales, es conveniente declararlas todas en un único módulo agrupadas por funciones y dar al módulo un nombre significativo que indique su finalidad, como Public.bas.

Una práctica de codificación correcta es escribir código modular siempre que sea posible. Por ejemplo, si la aplicación muestra un cuadro de diálogo, coloque todos los controles y el código necesario para ejecutar la tarea del diálogo en un único formulario. Esto ayuda a tener el código de la aplicación organizado en componentes útiles y minimiza la sobrecarga en tiempo de ejecución.

A excepción de las variables globales (que no se deberían pasar como parámetros), los procedimientos y funciones deben operar sólo sobre los objetos que se les pasan. Las variables globales que se usan en los procedimientos deben estar identificadas en la sección ***Declaraciones*** al principio del procedimiento.

Usar diversas variables con el mismo nombre

Si hay variables públicas en módulos distintos que comparten el mismo nombre, es posible diferenciarlas en el código haciendo referencia al módulo y a los nombres de variable. Por ejemplo, si hay una variable pública de tipo **Integer**, `intX` , declarada en Form1 y en Module1, puede hacer referencia a ambas como `Module1.intX` y `Form1.intX` para obtener los valores correctos:

```
Declaración en Module1:
    Public intX As Integer      ' Declara intX en Module1.
    Sub Test()
       intX = 1         ' Establece el valor de la variable intX en Module1.
    End Sub
```

Declaración en el módulo de formulario.

```
Public intX As Integer     ' Declara la variable intX del formulario.
Sub Test()
   intX = 2         ' Establece el valor de la variable intX del formulario.
End Sub
```

Respectivamente cada uno de los procedimientos de evento Click de los botones de comando de la aplicación llamará al procedimiento Test adecuado y utiliza **MsgBox** para mostrar los valores de las variables.

```
Private Sub Command1_Click()
   Module1.Test               ' Llama a Test en Module1.
   MsgBox Module1.intX         ' Muestra intX de Module1.
End Sub
Private Sub Command3_Click()
   Test                     ' Llama a Test en Form1.
   MsgBox intX               ' Muestra intX de Form1.
End Sub
```

Observe que en el procedimiento de evento Click del segundo botón de comando no necesita especificar `Form1.Test` cuando llame al procedimiento Test del formulario ni a `Form1.intX` cuando llame al valor de la variable **Integer** del formulario. Si hay varios procedimientos y variables con el mismo nombre, Visual Basic toma el valor de la variable más local que, en este caso, es la variable de Form1.

Variables públicas frente a locales

También puede tener una variable con el mismo nombre en alcances distintos. Por ejemplo, podría tener una variable pública llamada `Temp` y, en un procedimiento, declarar una variable local llamada `Temp`. Las referencias al nombre `Temp` dentro del procedimiento tendrían acceso a la variable local y las referencias a `Temp` fuera del procedimiento tendrían acceso a la variable pública. Se puede tener acceso a la variable a nivel de módulo desde el procedimiento calificándola con el nombre del módulo. Observese el ejemplo:

```
Public Temp As Integer

Private Sub Form_Load()
   Temp = 1               ' Establece Form1.Temp a 1.
End Sub
Private Sub Command1_Click()
   Test
End Sub
Sub Test()
   Dim Temp As Integer
   Temp = 2               ' Temp tiene el valor 2.
   MsgBox Form1.Temp      ' Form1.Temp tiene el valor 1.
End Sub
```

En general, cuando las variables tienen el mismo nombre pero distinto alcance, la variable más local siempre *oculta* (es decir, tiene preferencia de acceso) a las variables menos locales. Así, si tiene también una variable a nivel de procedimiento llamada `Temp`, ensombrecería a la variable pública `Temp` de ese módulo.

Usar variables y procedimientos con el mismo nombre

Los nombres de las variables privadas y públicas a nivel de módulo también pueden entrar en conflicto con los nombres de los procedimientos. *Una variable del módulo no puede tener el mismo nombre que los procedimientos o tipos definidos en el módulo. Sin embargo, pueden tener el mismo nombre que los procedimientos, tipos o variables públicos definidos en otros módulos.* En este caso, cuando se tiene acceso a la variable desde otro módulo, debe calificarse con el nombre del módulo.

> **Aunque las reglas de ensombrecimiento descritas antes no son complejas, el ensombrecimiento puede confundir y conducir a errores sutiles en el código; en programación es conveniente asignar nombres distintos a las variables. En los módulos de formulario, procure usar nombres de variable que difieran de los nombres de los controles utilizados en esos formularios.**

Convertir tipos de datos

Funciones de conversión	Convierten una expresión en
Cbool	Boolean
Cbyte	Byte
Ccur	Currency
Cdate	Date
CDbl	Double
Cint	Integer
CLng	Long
CSng	Single
CStr	String
Cvar	Variant
CVErr	Error

Visual Basic proporciona varias funciones de conversión que puede usar para convertir valores en tipos de datos específicos. Por ejemplo, para convertir un valor a **Currency**, utilice la función **CCur**:

```
PagoPorSemana = CCur(horas * PagoPorHora)
```

Nota Los valores que se pasan a una función de conversión deben ser válidos para el tipo de dato de destino o se producirá un error. Por ejemplo, si intenta convertir un tipo **Long** en un **Integer**, el tipo **Long** debe estar en el intervalo válido del tipo de dato **Integer**.

Intercambiar cadenas y números

Puede asignar una cadena a una variable numérica si la cadena representa un valor numérico. También es posible asignar un valor numérico a una variable de cadena. Por ejemplo, coloque un botón de comando, un cuadro de texto y un cuadro de lista en un formulario. Escriba el código siguiente en el evento Click del botón de comando. Ejecute la aplicación y haga clic en el botón de comando.

```
Private Sub Command1_Click()
   Dim intX As Integer
   Dim strY As String
   strY = "100.23"
   intX = strY                ' Pasa la cadena a una variable numérica.
   List1.AddItem Cos(strY)    ' Agrega el coseno del número de la
                            ' cadena al cuadro de lista.
   strY = Cos(strY)          ' Pasa el coseno a la variable de cadena.
   Text1.Text = strY          ' Se imprime la variable de cadena en el
                            ' cuadro de texto.
End Sub
```

Visual Basic convertirá automáticamente las variables al tipo de dato apropiado. Debe tener cuidado cuando intercambie números y cadenas, ya que pasar un valor no numérico a una cadena producirá un error de tiempo de ejecución.

Constantes

Puede mejorar mucho la legibilidad del código y facilitar su mantenimiento si utiliza constantes. Una *constante* es un nombre significativo que sustituye a un número o una cadena que no varía.

Hay dos orígenes para las constantes:

- *Constantes intrínsecas* o *definidas por el sistema* proporcionadas por aplicaciones y controles.
- *Las constantes simbólicas* o *definidas por el usuario* se declaran mediante la instrucción **Const**.

La sintaxis para declarar una constante es la siguiente:

```
[Public|Private] Const nombreConstante[As tipo] = expresión
```

El argumento *nombreConstante* es un nombre simbólico válido (las reglas son las mismas que para crear nombres de variable) y *expresión* está compuesta por constantes y operadores de cadena o numéricos; sin embargo, no puede usar llamadas a funciones en *expresión*.

Una instrucción **Const** puede representar una cantidad matemática o de fecha y hora:

```
Const PI = 3.14159265358979
Public Const MAXPLANETAS As Integer = 9
Const FECHASALIDA = #1/1/95#
```

Se puede usar también la instrucción **Const** para definir constantes de cadena:

```
Public Const VERSION = "07.10.A"
Const NOMBRECLAVE = "Enigma"
```

Puede colocar más de una declaración de constante en una única línea si las separa con comas:

```
Public Const PI = 3.14, MAXPLANETAS = 9, POBMUNDIAL = 6E+09
```

A menudo, la expresión del lado derecho del signo igual (=) es un número o cadena literal, pero también puede ser una expresión que dé como resultado un número o una cadena (aunque la función no puede contener llamadas a funciones). Puede incluso definir constantes en términos de constantes previamente definidas:

```
Const PI = PI * 2
```

Una vez que defina las constantes, puede colocarlas en el código para hacerlo más legible. Por ejemplo:

```
Static SistemaSolar(1 To MAXPLANETAS)
If numPersonas > POBMUNDIAL Then Exit Sub
```

Alcance de las constantes definidas por el usuario

Una instrucción **Const** tiene igual alcance que una declaración de variable y se le aplican las mismas reglas:

- Para crear una constante que sólo exista en un procedimiento, declárela dentro del procedimiento.
- Para crear una constante disponible para todos los procedimientos de un módulo, pero no para el código que está fuera del módulo, declárela en la sección Declaraciones del módulo.

- Para crear una constante disponible en toda la aplicación, declare la constante en la sección Declaraciones de un módulo estándar y coloque delante de **Const** la palabra clave **Public**. *No se pueden declarar las constantes públicas en un módulo de clase o de formulario.*

Evitar referencias circulares

Como es posible definir constantes en términos de otras constantes, deberá tener cuidado para no establecer un *ciclo* o referencia circular entre dos o más constantes. Se produce un ciclo cuando se tienen dos o más constantes públicas, cada una de las cuales está definida en función de la otra. Por ejemplo:

```
' En Module 1:
Public Const conA = conB * 2       ' Disponible en toda la aplicación.
' En Module 2:
Public Const conB = conA / 2       ' Disponible en toda la aplicación.
```

Si se produce un ciclo, Visual Basic generará un error cuando intente ejecutar la aplicación. No puede ejecutar el código hasta que resuelva la referencia circular. Para evitar la creación de un ciclo, restrinja todas las constantes públicas a un único módulo o, al menos, al menor número posible de módulos.

Operadores

Tipo	Operación	Operador
Aritmético	Exponenciación	^
	Cambio de signo	**-**
	Multiplicación y división	***, /**
	División entera	\
	Resto de una división entera	**Mod**
	Suma y resta	**+, -**
Concatenación	Concatenar o enlazar	**&**
Relacional	Igual, distinto, menor, mayor, …	**=, <>, <,>,<=,>=**
Otros	Comparar dos espresiones de caracteres	**Like**
	Comparar dos referencias a objetos	**Is**
Lógico	Negación	**Not**
	And	**And**
	Or inclusiva	**Or**
	Or exclusiva	**Xor**
	Equivalencia (opuesto a Xor)	**Eqv**
	Implicación (Falso si primer operador verdadero y segundo falso)	**Imp**

\ (Operador)

Se utiliza para dividir dos números y obtener un resultado entero. Sintaxis**:**

```
resultado = número1\número2
```

La sintaxis del operador \ consta de las siguientes partes:

Parte	Descripción
resultado	Requerido; cualquier variable numérica.
número1	Requerido; cualquier expresión numérica.
número2	Requerido; cualquier expresión numérica.

Ejemplo:

```
Dim MiValor
MiValor = 11 \ 4    ' Devuelve 2.
MiValor = 9 \ 3       ' Devuelve 3.
MiValor = 100 \ 3    ' Devuelve 33.
```

Comentarios: Antes de efectuar la división se redondean las expresiones numéricas para convertirlas en expresiones tipo Byte, Integer o Long. Normalmente, el tipo de dato del *resultado* es tipo **Byte**, **Byte** tipo Variant, tipo **Integer** o **Integer** tipo Variant, tipo **Long** o **Long** tipo Variant, independientemente de si el *resultado* es un número entero o no. La parte fraccionaria se trunca. Sin embargo, si cualquiera de las expresiones es Null, *resultado* es **Null**. Toda expresión que sea Empty se considera como 0.

Mod (Operador)

Divide dos números y devuelve sólo el resto. Sintaxis

```
resultado = número1 Mod número2
```

La sintaxis del operador **Mod** consta de las siguientes partes:

Parte	**Descripción**
resultado	Requerido; cualquier variable numérica.
número1	Requerido; cualquier expresión numérica.
número2	Requerido; cualquier expresión numérica.

Comentarios. El operador de módulo, o resto, divide *número1* por *número2* (redondeando a enteros los números de signo flotante) y devuelve sólo el resto como *resultado*. Por ejemplo, en la siguiente expresión, A (que es el *resultado*) es igual a 5.

```
A = 19 Mod 6.7
```

Generalmente, el tipo de dato de *resultado* es tipo Byte, **Byte** tipo variant, tipo Integer, **Integer** tipo variant, tipo Long o tipo Variant que contiene un tipo **Long**, independientemente de si el *resultado* es un número entero o no. La parte fraccionaria se trunca. Sin embargo, si cualquiera de las expresiones es Null, el *resultado* es también **Null**. Toda expresión Empty se considera como 0.

& (Operador)

Se utiliza para forzar la concatenación de las cadenas de dos expresiones. Sintaxis**:**

```
resultado = expresión1 & expresión2
```

La sintaxis del operador **&** consta de las siguientes partes:

Parte	**Descripción**
resultado	Requerido; cualquier variable tipo String o Variant.
expresión1	Requerido; cualquier expresión.
expresión2	Requerido; cualquier expresión.

Ejemplo:

```
Dim MyStr
MyStr = "Hola" & " mundo"            ' Devuelve "Hola mundo".
```

```
MyStr = "Prueba " & 123 & " Prueba"   ' Devuelve "Prueba 123 Prueba".
```

Comentarios: Si una *expresión* no es una cadena de caracteres, se convierte en un tipo **String** tipo variant. El tipo de dato del *resultado* es **String** si ambas expresiones son expresiones de cadena; de lo contrario, el *resultado* es una **String** tipo variant. Si ambas expresiones son **Null**, el *resultado* es también **Null**. Sin embargo, si sólo una *expresión* es **Null**, esa expresión se considera como una cadena de longitud cero ("") al concatenarse con la otra expresión. Cualquier expresión **Empty** se considera también una cadena de longitud cero.

Operadores de comparación

Se utilizan para comparar expresiones. Sintaxis:

```
resultado = expresión1 operadorcomparación expresión2
resultado = objeto1 Is objeto2
resultado = cadena Like patrón
```

Los operadores de comparación constan de las siguientes partes:

Parte	**Descripción**
resultado	Requerido; cualquier variable numérica.
expresión	Requerido; cualquier expresión.
operadorcomparación	Requerido; cualquier operador de comparación.
objeto	Requerido; cualquier nombre de objeto.
cadena	Requerido; cualquier expresión de cadena.
patrón	Requerido; cualquier expresión de cadena o intervalo de caracteres.

Comentarios: Esta tabla contiene una lista de los operadores de comparación y de las condiciones que determinan si el *resultado* es **True**, **False** o Null:

Operador	**True si**	**False si**	**Null si**
< (Menor que)	*expresión1 < expresión2*	*expresión1 >= expresión2*	*expresión1* o *expresión2* = **Null**
<= (Menor o igual que)	*expresión1 <= expresión2*	*expresión1 > expresión2*	*expresión1* o *expresión2* = **Null**
> (Mayor que)	*expresión1 > expresión2*	*expresión1 <= expresión2*	*expresión1* o *expresión2* = **Null**
>= (Mayor o igual que)	*expresión1 >= expresión2*	*expresión1 < expresión2*	*expresión1* o *expresión2* = **Null**
= (Igual a)	*expresión1 = expresión2*	*expresión1 <> expresión2*	*expresión1* o *expresión2* = **Null**
<> (Distinto de)	*expresión1 <> expresión2*	*expresión1 = expresión2*	*expresión1* o *expresión2* = **Null**

Nota Los operadores **Is** y **Like** tienen funciones de comparación específicas que difieren de las de los operadores incluidos en la siguiente tabla.

Cuando se comparan dos expresiones, puede ser difícil determinar si éstas se comparan como números o como cadenas de caracteres. En la siguiente tabla se muestra el modo en que se comparan expresiones y cuál es el resultado cuando cualquiera de las expresiones no es del tipo Variant:

Si	**Entonces**
Ambas expresiones son de tipos de datos numéricos (Byte, Boolean, Integer, Long, Single, Double, Date,	Se efectúa una comparación numérica.

Currency o Decimal)	
Ambas expresiones son de tipo String	Se efectúa una comparación de cadenas.
Una expresión es de tipo de datos numérico y la otra es un tipo Variant que es, o puede ser, un número	Se efectúa una comparación numérica.
Una expresión es de tipo de datos numérico y la otra es un tipo Variant de cadena de caracteres que no se puede convertir en un número	Provoca un error de `Error de tipos`.
Una expresión es de tipo String y la otra es cualquier tipo Variant, con excepción de Null	Se efectúa una comparación de cadenas.
Una expresión es Empty y la otra es del tipo de datos numérico	Se efectúa una comparación numérica, usando 0 como la expresión Empty.
Una expresión es Empty y la otra es del tipo String	Se efectúa una comparación de cadenas, utilizando una cadena de caracteres de longitud cero ("") como la expresión Empty.

Si tanto *expresión1* como *expresión2* son del tipo **Variant**, el tipo subyacente de las mismas es el que determina la manera en que se comparan. En la siguiente tabla se muestra el modo en que se comparan las expresiones y cuál es el resultado de la operación de acuerdo con el tipo subyacente de **Variant**:

Si	**Entonces**
Ambas expresiones tipo Variant son numéricas	Se efectúa una comparación numérica.
Ambas expresiones tipo Variant son cadenas de caracteres	Se efectúa una comparación de cadenas.
Una expresión tipo Variant es numérica y la otra es una cadena de caracteres	La expresión numérica es menor que la expresión de cadena.
Una expresión tipo Variant es Empty y la otra es numérica	Se efectúa una comparación numérica, usando 0 como la expresión Empty.
Una expresión tipo Variant es Empty y la otra es una cadena de caracteres	Se efectúa una comparación de cadenas, utilizando una cadena de caracteres de longitud cero ("") como la expresión Empty.
Ambas expresiones tipo Variant son Empty	Las expresiones son iguales.

Cuando se compara un tipo **Single** con un tipo **Double**, éste se redondea al grado de precisión del tipo **Single**.

Si una expresión tipo **Currency** se compara con una de tipo **Single** o **Double**, ésta se convierte al tipo **Currency**. De igual forma, cuando un tipo **Decimal** se compara con un tipo **Single** o **Double**, el tipo **Single** o el **Double** se convierten a tipo **Decimal**. Para el tipo **Currency**, cualquier valor fraccionario menor que 0,0001 se pierde; para el tipo **Decimal**, cualquier valor fraccionario menor que 1E-28 se pierde y puede ocurrir un error de desbordamiento. Perder valores fraccionarios puede hacer que dos valores se comparen como iguales cuando en realidad no lo son.

And (Operador)

Se utiliza para efectuar una conjunción lógica de dos expresiones. **Sintaxis:**

```
resultado = expresión1 And expresión2
```

La sintaxis del operador **And** consta de las siguientes partes:

Parte	Descripción
resultado	Requerido; cualquier variable numérica.
expresión1	Requerido; cualquier expresión.
expresión2	Requerido; cualquier expresión.

Comentarios: Si y sólo si ambas expresiones se evalúan como **True**, el *resultado* es **True**. Si cualquiera de las expresiones es **False**, el *resultado* es **False.** La siguiente tabla ilustra cómo se determina el *resultado*:

Si expresión1 es	**y expresión2 es**	**el resultado es**
True	True	True
True	False	False
False	True	False
False	False	False
False	Null	False
True	Null	Null
Null	True	Null
Null	False	False
Null	Null	Null

El operador **And** ejecuta también una comparación bit a bit para identificar los bits de dos expresiones numéricas que tienen la misma posición y establece el bit correspondiente en el *resultado* según la siguiente tabla de decisión lógica:

Si el bit en *expresión1* es	Y el bit en *expresión2* es	El *resultado* es
0	0	0
0	1	0
1	0	0
1	1	

Or (Operador)

Se utiliza para ejecutar una disyunción lógica sobre dos expresiones. **Sintaxis:**

```
resultado = expresión1 Or expresión2
```

La sintaxis del operador **Or** consta de las siguientes partes:

Parte	**Descripción**
Resultado	Requerido; cualquier variable numérica.
expresión1	Requerido; cualquier expresión.
expresión2	Requerido; cualquier expresión.

Comentarios: Si cualquiera de las expresiones, o ambas, es **True**, el *resultado* es **True**. La siguiente tabla indica cómo se determina el *resultado*:

Si *expresión1* **es**	**Y** *expresión2* **es**	**El** *resultado* **es**
True	**True**	**True**
True	**False**	**True**
True	Null	**True**

False	**True**	**True**
False	**False**	**False**
False	**Null**	**Null**
Null	**True**	**True**
Null	**False**	**Null**
Null	**Null**	**Null**

El operador **Or** ejecuta una comparación bit a bit para identificar los bits de dos expresiones numéricas que tienen la misma posición y establece el bit correspondiente en el *resultado* según la siguiente tabla de decisión lógica:

Si bit en *expresión1* **es**	**Si bit en** *expresión2* **es**	**El** *resultado* **es**
0	0	0
0	1	1
1	0	1
1	1	

Xor (Operador)

Se utiliza para realizar una exclusión lógica entre dos expresiones. **Sintaxis**

```
[resultado =] expresión1 Xor expresión2
```

La sintaxis del operador **Xor** consta de las siguientes partes:

Parte	Descripción
resultado	Requerido; cualquier variable numérica.
expresión1	Requerido; cualquier expresión.
expresión2	Requerido; cualquier expresión.

Comentarios: Si una y sólo una de las expresiones es **True**, el *resultado* es **True**. Sin embargo, si cualquiera de las expresiones es Null, el *resultado* es también **Null**. Cuando ninguna de las expresiones es **Null**, el *resultado* se determina de acuerdo con la siguiente tabla:

Si expresión1 **es**	**Y** expresión2 **es**	**El** resultado **es**
True	**True**	**False**
True	**False**	**True**
False	**True**	**True**
False	**False**	**False**

El operador **Xor** funciona como operador lógico y bit a bit. Ejecuta una comparación bit a bit para identificar los bits de dos expresiones utilizando lógica de O exclusivo para obtener el resultado, según la siguiente tabla de decisión lógica:

Si bit en *expresión1* **es**	**Y bit en e***xpresión2* **es**	**El** *result***ado es**
0	0	0
0	1	1
1	0	1
1	1	0

Like (Operador)

Se utiliza para comparar dos cadenas de caracteres. **Sintaxis**

```
resultado = cadena Like patrón
```

La sintaxis del operador **Like** consta de las siguientes partes:

Parte	Descripción
resultado	Requerido; cualquier variable numérica.
cadena	Requerido; cualquier expresión de cadena.
patrón	Requerido; cualquier expresión de cadena que satisface las convenciones de coincidencia de patrones descritas en Comentarios.

Comentarios: Si *cadena* coincide con *patrón*, el *resultado* es **True**; si no coincide, el *resultado* es **False**. Si *cadena* o *patrón* es Null, el *resultado* es también **Null**.

El comportamiento del operador **Like** depende de la instrucción **Option Compare**. El método predeterminado de comparación de cadenas para cada módulo es **Option Compare Binary**.

Option Compare Binary da como resultado comparaciones de cadenas basadas en el criterio de ordenación derivado de las representaciones binarias internas de los caracteres. En Microsoft Windows, el criterio de ordenación depende de la página de código. En el siguiente ejemplo se ilustra un criterio de ordenación binaria típico:

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

Option Compare Text da como resultado comparaciones de cadenas basadas en el criterio de ordenación determinado por la configuración regional de su sistema. Los mismos caracteres del ejemplo anterior, ordenados con la opción **Option Compare Text**, aparecen en el siguiente orden:

```
(A=a) < (À=à) < (B=b) < (E=e) < (Ê=ê) < (Z=z) < (Ø=ø)
```

La función integrada de búsqueda de coincidencia de patrones ofrece una herramienta versátil para efectuar comparaciones de cadenas. Las características de esta función permiten el empleo de caracteres comodín, listas de caracteres o intervalos de caracteres en cualquier combinación para hallar coincidencias en cadenas. En la siguiente tabla se indican los caracteres que se pueden poner en *patrón* y con qué coinciden los mismos:

Caracteres en *pattern*	**Coincidencias en** *string*
?	Un carácter cualquiera.
*****	Cero o más caracteres.
#	Un dígito cualquiera (0–9).
[*listacaracteres*]	Un carácter cualquiera de *listacaracteres*.
[**!***listacaracteres*]	Un carácter cualquiera no incluido en *listacaracteres*.

Se puede utilizar un grupo de uno o más caracteres (*listacaracteres*) entre corchetes (**[ ]**) para establecer una coincidencia con un carácter cualquiera de *cadena*; el grupo puede incluir casi cualquier código de carácter, incluyendo dígitos.

Nota Los caracteres especiales corchete de apertura (**[**), interrogación (**?**), signo de número (**#**) y asterisco (*****) se pueden utilizar para establecer una coincidencia con sí mismos sólo si van entre corchetes. El corchete de cierre (**]**) no se puede utilizar en un grupo para establecer una coincidencia con sí mismo, pero sí se puede utilizar fuera de un grupo, como carácter independiente.

Puede especificar un intervalo de caracteres en *listacaracteres* colocando un guión (–) para separar los límites inferior y superior del intervalo. Por ejemplo, la secuencia [`A-Z`] en *patrón* permite hallar una coincidencia si en la posición correspondiente de *cadena* hay un carácter en mayúsculas cualquiera, comprendido en el intervalo de la A a la Z. Se pueden incluir múltiples intervalos entre corchetes, sin necesidad de delimitadores.

El significado del intervalo especificado depende de la ordenación de caracteres válida en tiempo de ejecución (determinado por **Option Compare** y la configuración regional del sistema dónde está ejecutándose el código). Si se utiliza el ejemplo con **Option Compare Binary**, en el intervalo `[A–E]` coinciden A, B y E. Con **Option Compare Text**, en `[A–E]` coinciden A, a, À, à, B, b, E, e. Ê y ê no se incluyen entre las coincidencias porque los caracteres acentuados se encuentran después de los no acentuados en el criterio de ordenación.

Otras reglas importantes para efectuar coincidencias de patrones son las siguientes:

- Una exclamación (**!**) al comienzo de *listacaracteres* significa que hay coincidencias, dentro de la *cadena* para cualquier carácter excepto los incluidos en *listacaracteres*. Si no se encierra entre corchetes, la exclamación coincide consigo misma.
- El guión (–) puede aparecer tanto al comienzo (después de la exclamación, si se emplea) o al final de *listacaracteres* para coincidir consigo mismo. En cualquier otro lugar, el guión sólo se puede utilizar para identificar un intervalo de caracteres.
- Cuando se especifica un intervalo de caracteres, éstos deben aparecer en orden ascendente (de menor a mayor). `[A-Z]` es un patrón válido, pero `[Z-A]` no lo es.
- La secuencia de caracteres `[]` se considera una cadena de caracteres de longitud cero ("").

El alfabeto de algunos idiomas incluye caracteres especiales que en realidad representan dos caracteres distintos. Por ejemplo, varios idiomas emplean el carácter "æ" para representar los caracteres "a" y "e" cuando aparecen juntos. El operador **Like** reconoce que el carácter especial único y los dos caracteres individuales son equivalentes.

Cuando se especifica en la configuración regional del sistema un idioma que utiliza uno de estos caracteres especiales, al ocurrir el carácter especial en *patrón* o *cadena*, coincide con la secuencia equivalente de 2 caracteres en la otra cadena. Igualmente, un único carácter especial en *patrón* incluido entre corchetes (en solitario, en una lista o en un intervalo) coincide con la secuencia de 2 caracteres equivalente en *cadena*.

Is (Operador)

Se utiliza para comparar dos variables de referencia de objeto. Sintaxis:

```
resultado = objeto1 Is objeto2
```

La sintaxis del operador **Is** consta de las siguientes partes:

Parte	Descripción
resultado	Requerido; cualquier variable numérica.
objeto1	Requerido; cualquier nombre de objeto.
objeto2	Requerido; cualquier nombre de objeto.

Comentarios: Si tanto *objeto1* como *objeto2* se refieren al mismo objeto, el *resultado* es **True**; si no, el *resultado* es **False**. Hay varias maneras de hacer que dos variables se refieran al mismo objeto.

En el siguiente ejemplo, se ha definido A de modo que se refiera al mismo objeto que B:

```
Set A = B
```

En el siguiente ejemplo se hace que A y B se refieran al mismo objeto que C:

```
Set A = C
```

```
Set B = C
```

Cuadros de diálogo modales y no modales

Los cuadros de diálogo son un tipo especializado de objeto de formulario que se puede crear de tres maneras:

- Los cuadros de diálogo *predefinidos* se pueden crear desde el código mediante las funciones **MsgBox** o **InputBox**.
- Los cuadros de diálogo *personalizados* se pueden crear con un formulario estándar o si personaliza un cuadro de diálogo existente.
- Los cuadros de diálogo *estándar*, como **Imprimir** y **Abrir archivo**, se pueden crear con el control de diálogo común.

Como la mayoría de los cuadros de diálogo requieren la acción del usuario, suelen presentarse como cuadros de diálogo modales.

- Un cuadro de diálogo *modal* debe cerrarse (ocultar o descargar) antes de poder continuar trabajando con el resto de la aplicación. Por ejemplo, un cuadro de diálogo es modal si requiere que haga clic en **Aceptar** o en **Cancelar** antes de poder cambiar a otro formulario o cuadro de diálogo.
- Los cuadros de diálogo *no modales* permiten cambiar el enfoque entre el cuadro de diálogo y otro formulario sin tener que cerrar el cuadro de diálogo. Puede continuar trabajando en cualquier otra parte de la aplicación activa mientras se presenta el cuadro de diálogo. Los cuadros de diálogo no modales son escasos: normalmente se muestra un cuadro de diálogo porque es necesaria una respuesta antes de que pueda continuar la aplicación. El cuadro de diálogo **Buscar** del menú **Edición** de Visual Basic es un ejemplo de cuadro de diálogo no modal. Utilice los cuadros de diálogo no modales para mostrar comandos o información que se usen con frecuencia.

Cuadros de dialogo predefinidos

La forma más fácil de solicitar un dato a del usuario o de visualizar un resultado o un mensaje, es utilizando los cuadros de diálogo que Visual Basic provee para estos propósitos.

Pedir datos al usuario con InputBox

Sintaxis:

InputBox(mensaje,[titulo][,*default*][,posx],[poxy])

Muestra un mensaje en un cuadro de diálogo, espera que el usuario escriba un texto o haga clic en un botón y devuelve un tipo String con el contenido del cuadro de texto. En las aplicaciones basadas en Windows, se usan los cuadros de diálogo para pedir al usuario que especifique datos necesarios para que la aplicación pueda continuar o para mostrar información al usuario.

La sintaxis de la función **InputBox** consta de estos argumentos:

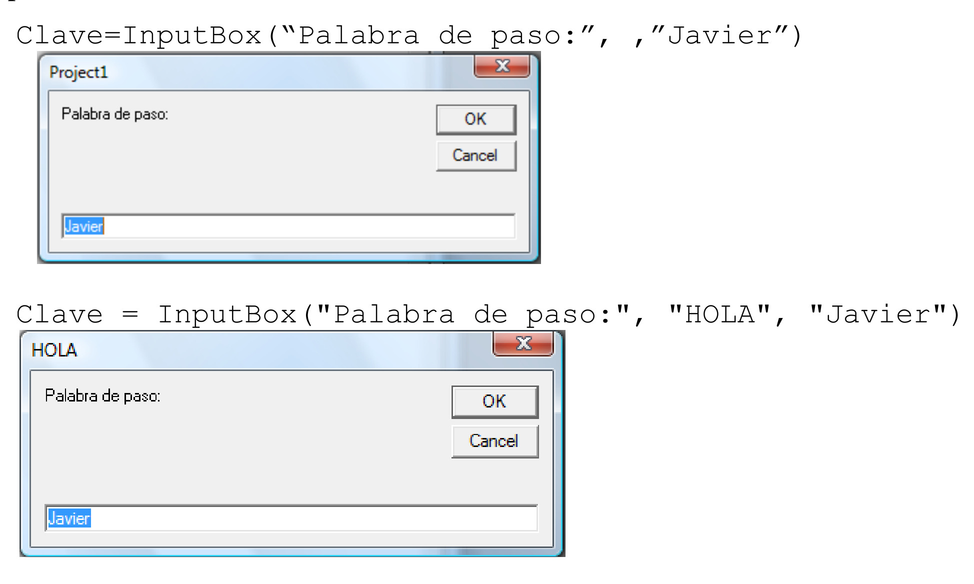
Parte	Descripción
mensaje	Requerido. Expresión de cadena que se muestra como mensaje en el cuadro de diálogo. La longitud máxima de **mensaje** es de aproximadamente 1024 caracteres, según el ancho de los

Parte	Descripción
	caracteres utilizados. Si **mensaje** consta de más de una línea, puede separarlos utilizando un carácter de retorno de carro (**Chr(**13**)**), un carácter de avance de línea (**Chr(**10**)**) o una combinación de los caracteres de retorno de carro-avance de línea (**Chr(**13**)** y **Chr(**10**)**) entre cada línea y la siguiente.
titulo	Opcional. Expresión de cadena que se muestra en la barra de título del cuadro de diálogo. Si omite ***titulo***, en la barra de título se coloca el nombre de la aplicación.
default	Opcional. Expresión de cadena que se muestra en el cuadro de texto como respuesta predeterminada cuando no se suministra una cadena. Si omite ***default***, se muestra el cuadro de texto vacío.
xpos	Opcional. Expresión numérica que especifica, en twips, la distancia en sentido horizontal entre el borde izquierdo del cuadro de diálogo y el borde izquierdo de la pantalla. Si se omite ***xpos***, el cuadro de diálogo se centra horizontalmente.
ypos	Opcional. Expresión numérica que especifica, en twips, la distancia en sentido vertical entre el borde superior del cuadro de diálogo y el borde superior de la pantalla. Si se omite ***ypos***, el cuadro de diálogo se coloca a aproximadamente un tercio de la altura de la pantalla, desde el borde superior de la misma.

Comentarios: Si el usuario hace clic en **Aceptar** o presiona Entrar, la función InputBox devuelve lo que haya en el cuadro de texto. Si el usuario hace clic en **Cancelar**, la función devuelve una cadena de caracteres de longitud cero (""). InputBox devuelve un dato de tipo Variant (de VarType igual a 8–String).

Ejemplo:

```
Clave=InputBox(“Palabra de paso:”, ,”Javier”)
```




```
Clave = InputBox("Palabra de paso:", "HOLA", "Javier")
```

Visualizar datos con MsgBox

Sintaxis:

```
MsgBox mensaje[, botones][,titulo]
```

Muestra un mensaje en un cuadro de diálogo, espera a que el usuario haga clic en un botón y devuelve un tipo **Integer** correspondiente al botón elegido por el usuario.

La sintaxis de la función **MsgBox** consta de estos argumentos con nombre:

Parte	**Descripción**
mensaje	Requerido. Expresión de cadena que representa el mensaje en el cuadro de diálogo. La longitud máxima de *mensaje* es de aproximadamente 1024 caracteres, según el ancho de los caracteres utilizados. Si *mensaje* consta de más de una línea, puede separarlos utilizando un carácter de retorno de carro (Chr(13)) o un carácter de avance de línea (Chr(10)), o una combinación de caracteres de retorno de carro – avance de línea (Chr(13) y Chr(10)) entre cada línea y la siguiente.
botones	Opcional. Expresión numérica que corresponde a la suma de los valores que especifican el número y el tipo de los *botones* que se pretenden mostrar, el estilo de icono que se va a utilizar, la identidad del botón predeterminado y la modalidad del cuadro de mensajes. Si se omite este argumento, el valor predeterminado para *botones* es 0.
titulo	Opcional. Expresión de cadena que se muestra en la barra de título del cuadro de diálogo. Si se omite *titulo*, en la barra de título se coloca el nombre de la aplicación.

Valores

El argumento ***botones*** tiene estos valores:

Constante	**Valor**	**Descripción**
VbOKOnly	0	Muestra solamente el botón **Aceptar**.
VbOKCancel	1	Muestra los botones **Aceptar** y **Cancelar**.
VbAbortRetryIgnore	2	Muestra los botones **Anular**, **Reintentar** e **Ignorar**.
VbYesNoCancel	3	Muestra los botones **Sí**, **No** y **Cancelar**.
VbYesNo	4	Muestra los botones **Sí** y **No**.
VbRetryCancel	5	Muestra los botones **Reintentar** y **Cancelar**.
VbCritical	16	Muestra el icono de **mensaje crítico**.
VbQuestion	32	Muestra el icono de **pregunta de advertencia**.
VbExclamation	48	Muestra el icono de **mensaje de advertencia**.
VbInformation	64	Muestra el icono de **mensaje de información**.
VbDefaultButton1	0	El primer botón es el predeterminado.
VbDefaultButton2	256	El segundo botón es el predeterminado.
VbDefaultButton3	512	El tercer botón es el predeterminado.
VbDefaultButton4	768	El cuarto botón es el predeterminado.
VbApplicationModal	0	Aplicación modal; el usuario debe responder al cuadro de mensajes antes de poder seguir trabajando en la aplicación actual.
VbSystemModal	4096	Sistema modal; se suspenden todas las aplicaciones hasta que el usuario responda al cuadro de mensajes.
VbMsgBoxHelpButton	16384	Agrega el botón Ayuda al cuadro de mensaje.
VbMsgBoxSetForeground	65536	Especifica la ventana del cuadro de mensaje como la ventana de primer plano.

VbMsgBoxRight	524288	El texto se alinea a la derecha.
VbMsgBoxRtlReading	1048576	Especifica que el texto debe aparecer para ser leído de derecha a izquierda en sistemas hebreo y árabe.

El primer grupo de valores (0 a 5) describe el número y el tipo de los botones mostrados en el cuadro de diálogo; el segundo grupo (16, 32, 48, 64) describe el estilo del icono, el tercer grupo (0, 256, 512) determina el botón predeterminado y el cuarto grupo (0, 4096) determina la modalidad del cuadro de mensajes. Cuando se suman números para obtener el valor final del argumento *botones*, se utiliza solamente un número de cada grupo.

Nota Estas constantes las especifica Visual Basic for Applications. Por tanto, el nombre de las mismas puede utilizarse en cualquier lugar del código en vez de sus valores reales.

Valores devueltos

El valor retornado por la función MsgBox indica qué botón se ha pulsado.

Constante	Valor	Descripción
vbOK	1	Aceptar
vbCancel	2	Cancelar
vbAbort	3	Anular
vbRetry	4	Reintentar
vbIgnore	5	Ignorar
vbYes	6	Sí
vbNo	7	No

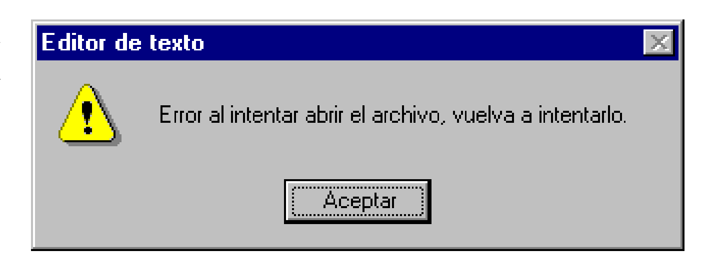
***Comentarios*:** Si el cuadro de diálogo cuenta con un botón **Cancelar**, presionar la tecla ESC tendrá el mismo efecto que hacer clic en este botón.

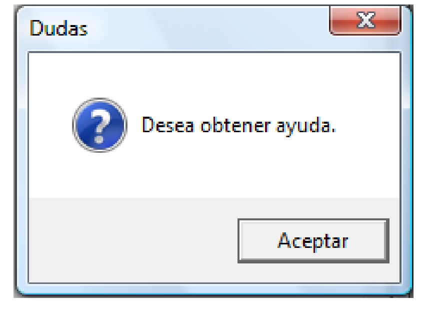
Se pueden proporcionar tres elementos informativos, o argumentos, a la función **MsgBox**:

- el texto del mensaje,
- una constante (valor numérico) para determinar el estilo del cuadro de diálogo y
- un título.

Este cuadro de diálogo aparece al invocar la función **MsgBox** en el código. El código para mostrarlo:

```
MsgBox "Error al intentar abrir el archivo, vuelva a intentarlo.", vbExclamation, "Editor de texto"
```





Otro ejemplo:

```
MsgBox "Desea obtener ayuda.", vbQuestion, "Dudas"
```

Existen estilos con diversas combinaciones de botones e iconos para facilitar la creación de cuadros de diálogo.

Procedimientos Function

Visual Basic incluye funciones incorporadas o intrínsecas, como **Sqr**, **Cos** o **Chr**. Además, puede usar la instrucción **Function** para escribir sus propios procedimientos **Function**.

Al igual que un procedimiento **Sub**, un procedimiento **Function** es un procedimiento diferente que puede tomar argumentos, realizar una serie de instrucciones y cambiar el valor de los argumentos. A diferencia de los procedimientos **Sub**, los procedimientos **Function** pueden devolver un valor al procedimiento que realiza la llamada.

Hay tres diferencias entre los procedimientos **Sub** y **Function**:

- Generalmente, se llama a una función incluyendo el nombre y los argumentos del procedimiento en la parte derecha de una instrucción o expresión mayor (valorRetorno = función()).
- Los procedimientos **Function** tienen tipos de datos, al igual que las variables. Esto determina el tipo del valor devuelto. (En ausencia de la cláusula **As**, el tipo es el tipo predeterminado **Variant**.)
- Se devuelve un valor asignándole al propio *nombreFuncion*. Cuando el procedimiento **Function** devuelve un valor, se puede convertir en parte de una expresión mayor.

La sintaxis de un procedimiento **Function** es la siguiente:

```
[Private|Public][Static]Function nombreFuncion(lista_ parámetros) [As tipo]
 instrucciones
 [nombre=expresión]
 [Exit Function]
 [instrucciones]
 [nombre=expresión]
End Function
```

La sintaxis de la instrucción **Function** consta de las siguientes partes:

Parte	**Descripción**
Public	Opcional. Indica que el procedimiento **Function** es accesible para todos los demás procedimientos de todos los módulos. Si se utiliza en un módulo que contiene **Option Private**, el procedimiento no estará disponible fuera del proyecto.
Private	Opcional. Indica que el procedimiento **Function** sólo es accesible para otros procedimientos del módulo donde se declara.
Static	Opcional. Indica que las variables locales del procedimiento **Function** se conservan entre las distintas llamadas. El atributo **Static** no afecta a las variables que se declaran fuera de **Function**, incluso aunque se utilicen en el procedimiento.
NombreFunción	**Requerido**. Nombre del procedimiento **Function**; sigue las convenciones estándar de nombres de variables.
lista_ parámetros	Opcional. Es una lista de variables separadas por comas que se corresponden con los argumentos que se pasan al procedimiento **Function** cuando se llama. Las variables múltiples se separan por comas. *(se detalla a continuación su sintaxis especifica)*
tipo	Opcional. Tipo de datos del valor devuelto por el procedimiento **Function**; puede ser

Parte	Descripción
	Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (no admitido actualmente), Date, String, o (excepto longitud fija), Object, Variant, o cualquier tipo definido por el usuario.
instrucciones	Opcional. Cualquier grupo de instrucciones que se van a ejecutar dentro del procedimiento **Function**.
expresión	Opcional. Valor de retorno de **Function**. Este valor es almacenado en el propio *nombre* de la función, que actúa como variable dentro del cuerpo de la misma. Si no se efectúa esta asignación, el resultado devuelto será 0 (cero) si ésta es numérica, nulo ("") si la función es de caracteres, o vacío (Empty) si la función es Variant.
Exit Function	Opcional. Permite salir de una función. No es necesaria a no ser que se necesite retornar a la sentencia situada inmediatamente a continuación de la que efectuó la llamada antes de que la función finalice.
End Function	**Requerido.** Al igual que Exit Function, devuelve el control a la sentencia que se encuentra inmediatamente a continuación de la que efectuó la llamada, continuando de esta forma la ejecución del programa.

Llamar a procedimientos Function

Se llama a un procedimiento **Function,** que se ha escrito, de la misma forma que a las funciones incorporadas en Visual Basic (funciones intrínsecas, como **Abs**), es decir, utilizando su nombre en una expresión. Por ejemplo las instrucciones siguientes llaman a una función llamada Adec.

```
Print 10 * Adec                 'desde una función de impresion
X = Adec                        'asignando el valor devuelto a una variable
If Adec = 10 Then Debug.Print "Fuera del intervalo"
                                'desde una sentencia de control
X = OtraFunción(10 * Adec)      'como argumento de otra función
```

También es posible llamar a una función igual que se llama a un procedimiento **Sub**. Las instrucciones siguientes llaman a la misma función:

```
Call Year(Now)
Year Now
```

Cuando llama a una función de esta manera, Visual Basic desecha el valor devuelto.

Salir de un procedimiento Function

Se puede salir de una función desde una estructura de control. **Exit Function** puede aparecer tantas veces como se necesite, en cualquier parte del cuerpo de un procedimiento **Function**.

Exit Function es muy útiles cuando el procedimiento ha realizado todo lo que tenía que hacer y se quiere volver inmediatamente.

Para crear una funcion general nueva

Escriba el encabezado de un procedimiento en la ventana Código y presione ENTRAR. El encabezado del procedimiento puede ser tan simple como **Function** seguido de un nombre. Por ejemplo, puede especificar cualquiera de los siguientes:

```
Function Factorial ()
```

```
Function ObtenerCoord ()
```

Visual Basic responde completando la plantilla del nuevo procedimiento.

```
Function Factorial ()
End Function
```

Ejemplo de una función ya terminada:

```
Public Function Factorial(N AS Integer, F AS Long)
 If N=0 Then
      F=1
 Else
      F=1
      Do While N>0
           F=N*F
        N=N-1
      Loop
 End If
End Function
```

Implicaciones del paso de parámetros

Normalmente el código de un procedimiento necesita cierta información sobre el estado del programa para realizar su trabajo. Esta información consiste en variables que se pasan al procedimiento cuando se le llama. En la llamada cuando se pasa una variable a un procedimiento, se llama *argumento*.

Los argumentos de los procedimientos que escriba tienen el tipo de dato **Variant** de forma predeterminada. Sin embargo, puede declarar otros tipos de datos para los argumentos.

El argumento *lista_ parámetros* de la sintaxis de **Function o Sub** tiene la siguiente sintaxis y partes:

[ByVal | **ByRef**] [**ParamArray**] *nombre_variable*[**( )**] [**As** *tipo*] [**=** *valor_predeterminado*]

Parte	**Descripción**
ByVal	Opcional. Indica que el argumento se pasa por valor.
ByRef	Opcional. Indica que el argumento se pasa por referencia. **ByRef** es el valor predeterminado en Visual Basic.
ParamArray	Opcional. Sólo se utiliza como último argumento de *lista_argumentos* para indicar que el último argumento es una matriz **Optional** de elementos **Variant**. La palabra clave **ParamArray** le permite proporcionar un número arbitrario de argumentos. No puede utilizarse con **ByVal**, **ByRef**.
nombre_ variable	**Requerido**. Nombre de la variable que representa el argumento; sigue las convenciones estándar de nombres de variables.
tipo	Opcional. Tipo de datos del argumento pasado al procedimiento; puede ser **Byte**, **Boolean**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Decimal** (no compatible actualmente), **Date**, **String** (sólo longitud variable), **Object**, **Variant**, o un tipo de objeto específico. Si el parámetro no es **Optional**, se puede especificar también un tipo definido por el usuario.
*valor_*	Opcional. Cualquier constante o expresión constante. Sólo es válido para parámetros

Parte	Descripción
predeterminado	**Optional**. Si el tipo es **Object**, un valor predeterminado explícito sólo puede ser **Nothing**.

Por ejemplo, podría escribir una función que calculara el tercer lado, o hipotenusa, de un triángulo rectángulo, dados los valores de los otros dos lados:

```
Function Hipotenusa (A As Integer, B As Integer) As String
   Hipotenusa = Sqr(A ^ 2 + B ^ 2)
End Function
```

En su caso la llamada a la función podría hacerse como sigue:

```
Label1.Caption = Hipotenusa(CInt(Text1.Text), CInt(Text2.Text))
strX = Hipotenusa(Width, Height)
```

Observe los argumentos que son pasados a la función, los cuales serán recibidos por los parámetros. El primer argumento Text1.Text es recibido por el primer parámetro, A, *(observese que es convertido a entero con la función CInt()),* el segundo argumento Text2.Text es recibido por el segundo parámetro, ***B*** *(observese igualmente que es convertido a entero con la función CInt() ya que ambos parámetros en la función Hipotenusa son enteros y las cajas de texto contienen cadenas de caracteres).*

Otro ejemplo, la función siguiente acepta una cadena y un entero:

```
Function QueComer(DíaSemana As String, Hora As Integer) As String
   ' Devuelve el menú del almuerzo basándose en el día y la hora.
   If DíaSemana = "Viernes" then
      QuéComer = "Pescado"
   Else
      QuéComer = "Pollo"
   End If
   If Hora > 4 Then QuéComer = "Demasiado tarde"
End Function
```

Pasar argumentos por valor

Sólo se pasa una copia de la variable cuando se pasa un argumento por valor. Si el procedimiento cambia el valor, el cambio afecta sólo a la copia y no a la variable propiamente dicha. Utilice la palabra clave **ByVal** para indicar un argumento pasado por valor. Por ejemplo:

```
Sub Cuentas (ByVal intNumCuenta as Integer)
   ' Ponga aquí sus instrucciones.
End Sub
```

Pasar argumentos por referencia

Pasar argumentos por referencia le da al procedimiento acceso al contenido real de la variable en su ubicación de dirección de memoria. Como resultado, el procedimiento al que se ha pasado el valor de la variable se puede modificar de forma permanente. **La forma predeterminada de pasar valores en Visual Basic es por referencia.**

Si especifica el tipo de dato de un argumento que se pasa por referencia, debe pasar un valor de ese tipo para el argumento. Puede eludirlo si pasa una expresión en vez de un tipo de dato como argumento. Visual Basic evalúa la expresión y la pasa como el tipo requerido si puede.

La forma más sencilla de convertir una variable en una expresión es ponerla entre paréntesis. Por ejemplo, para pasar una variable declarada como entero a un procedimiento que espera una cadena como argumento, debería hacer lo siguiente:

```
Sub ProcedimientoQueLlama ()
   Dim intX As Integer
   intX = 12 * 3
   Foo(intX)
End Sub

Sub Foo(Bar As String)
   MsgBox Bar   'El valor de Bar es la cadena "36".
End Sub
```

Introducción a las estructuras de control

Las estructuras de control le permiten controlar el flujo de ejecución del programa. Si no se controla mediante instrucciones de control de flujo, la lógica del programa fluirá por las instrucciones de izquierda a derecha y de arriba a abajo. Aunque se pueden escribir algunos programas sencillos con un flujo unidireccional y aunque se puede controlar parte del flujo mediante operadores para regular la precedencia de las operaciones, **la mayor parte del poder y utilidad de un lenguaje de programación deriva de su capacidad de cambiar el orden de las instrucciones mediante estructuras y bucles.**

Estructuras de decisión

Los procedimientos de Visual Basic pueden probar condiciones y, dependiendo de los resultados de la prueba, realizar diferentes operaciones. Entre las estructuras de decisión que acepta Visual Basic se incluyen las siguientes:

- If...Then
- If...Then...Else
- Select Case

If...Then

Use la estructura **If...Then** para ejecutar una o más instrucciones basadas en una condición. Puede usar la sintaxis de una línea o un *bloque* de varias líneas.

Sintaxis 1:

```
If condición Then instrucción
```

Sintaxis 2:

```
If condición Then
 instrucciones
End If
```

Parte	Descripción
condición	Requerido. Uno o más de los siguientes dos tipos de expresiones:
	Una expresión numérica o expresión de cadena que puede ser evaluada como **True** o **False**. Si *condición* es Null, *condición* se considera **False**.
instrucciones	Opcional en formato de bloque; se requiere en formato de línea sencilla que no tenga una cláusula **Else**. Una o más instrucciones separadas por dos puntos ejecutados si la *condición* es **True**.

Condición normalmente es una comparación, pero puede ser cualquier expresión que dé como resultado un valor numérico. Visual Basic interpreta este valor como **True** o **False**; un valor numérico cero es **False** y se considera **True** cualquier valor numérico distinto de cero. Si *condición* es **True**, Visual Basic ejecuta todas las *instrucciones* que siguen a la palabra clave **Then**.

Puede usar la sintaxis de una línea o de varias líneas para ejecutar una instrucción basada en una condición.

Estos dos ejemplos son equivalentes:

```
If cualquierFecha < Now Then cualquierFecha = Now
```

O sino:

```
If cualquierFecha < Now Then
   cualquierFecha = Now
End If
```

Observe que el formato de una única línea de **If...Then** no utiliza la instrucción **End If**. Si desea ejecutar más de una línea de código cuando *condición* sea **True**, debe usar la sintaxis de bloque de varias líneas **If...Then...End If**.

```
If cualquierFecha < Now Then
   cualquierFecha = Now
   Timer1.Enabled = False   ' Desactiva el control Timer.
End If
```

If...Then...Else

Ejecuta condicionalmente un grupo de instrucciones, dependiendo del valor de una expresión. Sintaxis 3:

```
If condición Then [instrucciones][Else instrucciones_else]
```

Sintaxis 4:

```
If condición Then
        [instrucciones]
[Else
        [instrucciones_else]]
End If
```

La sintaxis de la instrucción **If...Then...Else** consta de tres partes:

Parte	**Descripción**
condición-n	Opcional. Igual que *condición*.
instrucciones_else	Opcional. Una o más instrucciones ejecutadas si la condición es True. **Instrucción:**Una unidad sintácticamente completa que expresa un tipo de acción, declaración o definición. Normalmente una instrucción tiene una sola línea aunque es posible utilizar dos puntos (:) para poner más de una instrucción en una línea. También se puede utilizar un carácter de continuación de línea (_) para continuar una sola línea lógica en una segunda línea física.

Observe que el formato de una única línea de **If...Then** no utiliza la instrucción **End If**. Si desea ejecutar más de una línea de código cuando *condición* sea **True**, debe usar la sintaxis de bloque de varias líneas **If...Then...End If**.

Ejemplos:

```
If num = 0 Then MsgBox ("cero") Else MsgBox ("no es cero")
```

Ejemplo:

```
If num = 0 Then
```

```
        MsgBox ("cero")
Else
       MsgBox ("no es cero")
End If
```

Puede contener sentencias anidadas de la siguiente forma:

```
If num = 0 Then
        MsgBox ("Es cero")
Else
        If num < 0 Then
             MsgBox ("Es negativo")
        Else
             MsgBox ("Es positivo")
        End If
End If
```

En formato de bloque puede utilizar la siguiente sintaxis:

```
If condición Then
        [instrucciones]
[ElseIf condición-n Then
        [instrucciones_elseif] ...
 [Else
        [instrucciones_else]]
End If
```

Parte	**Descripción**
instrucciones_elseif	Opcional. Una o más instrucciones ejecutadas si la *condición-n* asociada es **True**.

Ejemplo:

```
If num = 0 Then
        MsgBox ("Es cero")
ElseIf num < 0 Then
        MsgBox ("Es negativo")
Else
        MsgBox ("Es positivo")
End If
```

Utilice un bloque **If...Then...Else** para definir varios bloques de instrucciones, uno de los cuales se ejecutará:

```
If condición1 Then
        [bloque de instrucciones 1]
[ElseIf condición2 Then
        [bloque de instrucciones 2]]
[ElseIf condición3 Then
        [bloque de instrucciones 3]]
[Else
        [bloque de instrucciones n]]
End If
```

Visual Basic evalúa primero *condición1*:

- Si es **False**, Visual Basic procede a evaluar *condición2*, luego la *condición3* y así sucesivamente, hasta que encuentre una condición **True**. Si encuentra una condición **True**, Visual Basic ejecuta el bloque de instrucciones correspondientes y después ejecuta el código que sigue a **End If**.

- Opcionalmente, puede incluir un bloque de instrucciones **Else**, que Visual Basic ejecutará si ninguna de las condiciones es **True**.

If...Then...ElseIf es un caso especial de **If...Then...Else**. Observe que puede tener cualquier número de cláusulas **ElseIf** o ninguna. Puede incluir una cláusula **Else** sin tener en cuenta si tiene o no cláusulas **ElseIf**.

Ejemplo:

```
If letra = "a" Then
       MsgBox ("Es vocal")
ElseIf letra = "e" Then
       MsgBox ("Es vocal")
ElseIf letra = "i" Then
       MsgBox ("Es vocal")
ElseIf letra = "o" Then
       MsgBox ("Es vocal")
ElseIf letra = "u" Then
       MsgBox ("Es vocal")
Else
       MsgBox ("No es vocal")
End If
```

Otra forma de escribirla cuando es una sola línea seria:

```
If letra = "a" Then      MsgBox ("Es vocal")
ElseIf letra = "e" Then  MsgBox ("Es vocal")
ElseIf letra = "i" Then  MsgBox ("Es vocal")
ElseIf letra = "o" Then  MsgBox ("Es vocal")
ElseIf letra = "u" Then  MsgBox ("Es vocal")
Else   MsgBox ("No es vocal")
End If
```

Observe que siempre puede agregar más cláusulas **ElseIf** a la estructura **If...Then**. Sin embargo, esta sintaxis puede resultar tediosa de escribir cuando cada **ElseIf** compara la misma expresión con un valor distinto. Para estas situaciones, puede usar la estructura de decisión **Select Case**.

Select Case

Visual Basic proporciona la estructura **Select Case** como alternativa a **If...Then...Else** para ejecutar selectivamente un bloque de instrucciones entre varios bloques de instrucciones. La instrucción **Select Case** ofrece posibilidades similares a la instrucción **If...Then...Else**, pero hace que el código sea más legible cuando hay varias opciones.

La estructura **Select Case** funciona con una única expresión de prueba que se evalúa una vez solamente, al principio de la estructura. Visual Basic compara el resultado de esta expresión con los valores de cada **Case** de la estructura. Si hay una coincidencia, ejecuta el bloque de instrucciones asociado a ese **Case.**

Sintaxis:

```
Select Case expresión_prueba
[Case lista_expresion-n
       [instrucciones-n]] ...
[Case Else
       [instrucciones_else]]
End Select
```

Parte	**Descripción**

expresión_prueba	**Requerido**. Cualquier expresión o variable numérica o expresión de cadena.
lista_expresión-n	**Requerido** si aparece la palabra clave **Case**. Cada *lista_expresion* es una lista de uno o más valores. Si hay más de un valor en una lista, se separan los valores con comas. Lista delimitada por comas de una o más de las formas siguientes: *Expresión[, expresión]*... *expresión* **To** *expresión*, **Is** *expresión operador_de_comparación*. La palabra clave especifica un intervalo de valores. Si se utiliza la palabra clave **To**, el valor menor debe aparecer antes de **To**. Utilice la palabra clave **Is** con operadores de comparación (excepto **Is** y **Like**) para especificar un intervalo de valores. Si no se escribe, la palabra clave **Is** se insertará automáticamente.
instrucciones-n	Opcional. Una o más instrucciones ejecutadas si *expresión_prueba* coincide con cualquier parte de *lista_expresión-n.*
instrucciones_ else	Opcional. Una o más instrucciones que se ejecuten si *expresión_prueba* no coincide con nada de la cláusula **Case**.

Cada *bloque de instrucciones* contiene cero o más instrucciones. Si más de un **Case** coincide con la expresión de prueba, sólo se ejecutará el bloque de instrucciones asociado con la primera coincidencia. Visual Basic ejecuta las instrucciones de la cláusula (opcional) **Case Else** si ningún valor de la lista de expresiones coincide con la expresión de prueba.

Ejemplo:

```
Select Case letra
 Case "a"
       MsgBox ("Es vocal")
 Case "e"
       MsgBox ("Es vocal")
 Case "i"
       MsgBox ("Es vocal")
 Case "o"
       MsgBox ("Es vocal")
 Case "u"
       MsgBox ("Es vocal")
 Case Else
       MsgBox ("No es vocal")
End Select
```

O podría quedar también asi;

```
Select Case letra
 Case "a", "e", "i", "o", "u"
      MsgBox ("Es vocal")
 Case Else
      MsgBox ("No es vocal")
End Select
```

Ejemplo:

```
Select Case num
 Case 0 To 9     MsgBox ("es digito")
 Case Is < 0     MsgBox ("Es negativo")
 Case Is >= 10   MsgBox ("es positivo, no digito")
End Select
```

Observe que la estructura **Select Case** evalúa una expresión cada vez al principio de la estructura. Por el contrario, la estructura **If...Then...Else** puede evaluar una expresión diferente en cada instrucción **ElseIf**. Sólo puede sustituir una estructura **If...Then...Else** con una estructura **Select Case** si la instrucción **If** y cada instrucción **ElseIf** evalúa la misma expresión.

Nota: **Exit** es una instrucción que Saca de un bloque de código de **Do…Loop**, **For...Next**, **Function**, **Sub**. Sintaxis:

```
Exit Do
Exit For
Exit Function
Exit Property
Exit Sub
```

Estructuras de bucle

Las estructuras de bucle le permiten ejecutar una o más líneas de código repetidamente. Las estructuras de bucle que acepta Visual Basic son:

- While...Wend
- Do...Loop
- For...Next
- For Each...Next

While ...Wend

Ejecuta una serie de instrucciones mientras una condición dada sea **True**. Sintaxis:

```
While condición
      [intrucciones]
Wend
```

La sintaxis de la instrucción **While...Wend** consta de las siguientes partes:

Parte	**Descripción**
condición	Requerido. Expresión numérica o expresión de cadena cuyo valor es **True** o **False**. Si *condición* es Null, *condición* se considera **False**.
instrucciones	Opcional. Una o más instrucciones que se ejecutan mientras la condición es **True**.

Si *condición* es **True**, todas las instrucciones se ejecutan hasta que se encuentra la instrucción **Wend**. Después, el control vuelve a la instrucción **While** y se comprueba de nuevo *condición*. Si

condición es aún **True**, se repite el proceso. Si no es **True**, la ejecución se reanuda con la instrucción que sigue a la instrucción **Wend**.

Ejemplo:

```
While (num = 0)
    num = InputBox("Dame un numero", "Positivo,Negativo,Cero")
Wend
```

Los bucles **While...Wend** se pueden anidar a cualquier nivel. Cada **Wend** coincide con el **While** más reciente.

Sugerencia La instrucción **Do...Loop** proporciona una manera más flexible y estructurada de realizar los bucles.

Do...Loop

Utilice el bucle **Do** para ejecutar un bloque de instrucciones un número indefinido de veces, mientras la condición dada sea cierta, o hasta que una condición dada sea cierta. Hay algunas variantes en la instrucción **Do...Loop**, pero cada una evalúa una condición numérica para determinar si continúa la ejecución. Como ocurre con **If...Then**, la *condición* debe ser un valor o una expresión que dé como resultado **False** (cero) o **True** (distinto de cero).

La condición puede ser verificada antes o después de ejecutarse el conjunto de sentencias:

Sintaxis 1:

```
Do [{While | Until} condición]
       [instrucciones]
      [Exit Do]
       [instrucciones]
Loop
```

Sintaxis 2:

```
Do
     [instrucciones]
      [Exit Do]
     [instrucciones]
Loop [{While | Until} condición]
```

Instrucción	Descripción
Exit Do	Proporciona una manera de salir de una instrucción **Do...Loop**. Solamente se puede utilizar dentro de una instrucción **Do...Loop**. La instrucción **Exit Do** transfiere el control a la instrucción que sigue a la instrucción **Loop**. Cuando se utiliza con instrucciones anidadas **Do...Loop**, la instrucción **Exit Do** transfiere el control al bucle que está anidado un nivel por encima del bucle donde ocurre. Se puede utilizar cualquier número de instrucciones **Exit Do** ubicadas en cualquier lugar dentro de una estructura de control **Do…Loop**, para proporcionar una salida alternativa de un **Do…Loop**. La instrucción **Exit Do** se utiliza frecuentemente en la evaluación de alguna condición, por ejemplo, **If…Then**; en este caso, la instrucción **Exit Do** transfiere el control a la instrucción que sigue inmediatamente a la instrucción **Loop**.

	Cuando se utiliza con instrucciones anidadas **Do…Loop**, la instrucción **Exit Do** transfiere control al bucle que está anidado un nivel por encima del bucle donde ocurre.
While	Si se usa While, el bucle se puede ejecutar cualquier número de veces, siempre y cuando *condición* sea distinta de cero o **True**. Nunca se ejecutan las instrucciones si *condición* es **False**
Until	Si se usa Until, se repiten el bucle siempre y cuando *condición* sea **False** en vez de **True**. Nunca se ejecutan las instrucciones si *condición* es **true**

Cuando Visual Basic ejecuta el bucle **Do** de la sintaxis 1, primero evalúa *condición*.

1. Si *condición* es **False** (cero), se salta todas las instrucciones.
2. Si es **True** (distinto de cero), Visual Basic ejecuta las instrucciones, vuelve a la instrucción **Do While** y prueba la condición de nuevo.

Nunca se ejecutan las instrucciones si *condición* es *False* inicialmente (es decir entra cero o más veces). Ejemplo:

```
num = InputBox("Dame un numero", "Positivo,Negativo,Cero")
Do While (num = 0)
    num = InputBox("Dame un numero", "Positivo,Negativo,Cero")
Loop
```

Otro ejemplo:

```
contador = 1
Do While contador < 20
   Print "hola"
   contador = contador +1
Loop
```

Otra variante de la instrucción **Do...Loop**, *usando While*, es la que se muestra en la sintaxis 2, que ejecuta las instrucciones primero y prueba *condición* después de cada ejecución. Esta variación **garantiza al menos una ejecución de *instrucciones***. Ejemplo:

```
contador = 1
Do
     Print "hola"
     contador = contador +1
Loop While contador < 20
```

Hay dos variantes análogas a las dos sintaxis donde se usa While, solo que en estas se repiten el bucle siempre y cuando *condición* sea **False** en vez de **True**.

Hace el bucle cero o más veces	Hace el bucle al menos una vez
Do Until condición Instrucciones **Loop**	**Do** Instrucciones **Loop Until** condición

For...Next

Los bucles **Do** funcionan bien cuando no se sabe cuántas veces se necesitará ejecutar las instrucciones del bucle. Sin embargo, **cuando se sabe que se van a ejecutar las instrucciones un número determinado de veces**, es mejor elegir el bucle **For…Next**. A diferencia del bucle **Do**, el

bucle **For** utiliza una variable llamada contador que incrementa o reduce su valor en cada repetición del bucle. La sintaxis es la siguiente:

```
For contador = iniciar To finalizar [Step incremento]
      [instrucciones]
      [Exit For]
      [instrucciones]
Next [contador]
```

Los argumentos contador, iniciar, finalizar e incremento son todos numéricos.

La sintaxis de la instrucción **For...Next** consta de las siguientes partes:

Parte	**Descripción**
contador	Requerido. Variable numérica que se utiliza como contador de bucle. La variable no puede ser Booleana ni un elemento de matriz.
iniciar	Requerido. Valor inicial del *contador*.
finalizar	Requerido. Valor final del *contador*.
incremento	Opcional. Cantidad en la que cambia el *contador* cada vez que se ejecuta el bucle. Si no se especifica, el valor predeterminado de *incremento* es uno.
instrucciones	Opcional. Una o más instrucciones entre **For** y **Next** que se ejecutan un número especificado de veces.
Exit For	Proporciona una manera de salir de un bucle **For**. Sólo se puede utilizar en un bucle **For...Next** o **For Each...Next**. La instrucción **Exit For** transfiere el control a la instrucción que sigue a la instrucción **Next**. Cuando se utiliza con bucles anidados **For**, la instrucción **Exit For** transfiere el control al bucle que está anidado un nivel por encima del bucle donde ocurre.

Nota El argumento *incremento* puede ser positivo o negativo. Si *incremento* es positivo, *iniciar* debe ser menor o igual que *finalizar* o no se ejecutarán las instrucciones del bucle. Si *incremento* es negativo, *iniciar* debe ser mayor o igual que *finalizar* para que se ejecute el cuerpo del bucle. Si no se establece **Step**, el valor predeterminado de *incremento* es 1.

Valor	**El bucle se ejcuta si**
Positivo o 0	*contador* <= *fin*
Negativo	*contador* >= *fin*

Una vez que se inicia el bucle y se han ejecutado todas las instrucciones en el bucle, *incremento* se suma a *contador*. En este punto, las instrucciones del bucle se pueden ejecutar de nuevo (si se cumple la misma prueba que causó que el bucle se ejecutara inicialmente) o bien se sale del bucle y la ejecución continúa con la instrucción que sigue a la instrucción **Next**.

Sugerencia Cambiar el valor de *contador* mientras está dentro de un bucle hace difícil su lectura y depuración.

Al ejecutar el bucle For, Visual Basic:

1. Establece *contador* al mismo valor que *iniciar*.
2. Comprueba si *contador* es mayor que *finalizar*. Si lo es, Visual Basic sale del bucle.
3. (Si *incremento* es negativo, Visual Basic comprueba si *contador* es menor que *finalizar*.)

4. Ejecuta *instrucciones*.
5. Incrementa *contador* en 1 o en *instrucciones*, si se especificó.
6. Repite los pasos 2 a 4.

Este código imprime los nombres de todas las fuentes de pantalla disponibles:

```
Private Sub Form_Click ()
   Dim I As Integer
   For i = 0 To Screen.FontCount
      Print Screen.Fonts(i)
   Next
End Sub
```

```
Ejemplo:
     For contador = 1 To 20
         Print contador
     Next
```

Se pueden colocar en el bucle cualquier número de instrucciones **Exit For** como una manera alternativa de salir del mismo. La instrucción **Exit For**, que se utiliza a menudo en la evaluación de alguna condición (por ejemplo, **If...Then**), transfiere el control a la instrucción que sigue inmediatamente a la instrucción **Next**.

Se pueden anidar bucles **For...Next**, colocando un bucle **For...Next** dentro de otro. Para ello, proporcione a cada bucle un nombre de variable único como su *contador*. La siguiente construcción es correcta:

```
For I = 1 To 10
   For J = 1 To 10
      For K = 1 To 10
         Print I,J
      Next K
   Next J
Next I
```

Nota Si omite un *contador* en una instrucción **Next**, la ejecución continúa como si se hubiera incluido. Se produce un error si se encuentra una instrucción **Next** antes de su instrucción **For** correspondiente.

```
Private Sub Form_Click()
Dim Matriz(5) As String
Dim x As Integer
   For x = 1 To 5
      List1.AddItem InputBox("Dame un color:")
   Next x
End Sub
```

For Each...Next

El bucle **For Each...Next** es similar al bucle **For...Next**, pero repite un grupo de instrucciones por cada elemento de una colección de objetos o de una matriz en vez de repetir las instrucciones un número especificado de veces. Esto resulta especialmente útil si no sabe cuántos elementos hay en la colección. Sintaxis:

```
For Each elemento In grupo
      [instrucciones]
      [Exit For]
```

```
        [instrucciones]
    Next elemento
```

La sintaxis de la instrucción **For Each...Next** consta de las siguientes partes:

Parte	Descripción
elemento	Requerido. Variable que se utiliza para iterar por los elementos del conjunto o matriz. Para conjuntos, *elemento* solamente puede ser una variable del tipo Variant, una variable de objeto genérica o cualquier variable de objeto específica. Para matrices, *elemento* solamente puede ser una variable tipo **Variant**.
grupo	Requerido. Nombre de un conjunto de objetos o de una matriz (excepto una matriz de tipos definidos por el usuario).
instrucciones	Opcional. Una o más instrucciones que se ejecutan para cada elemento de un *grupo*.

La entrada al bloque **For Each** se produce si hay al menos un elemento en *grupo*. Una vez que se ha entrado en el bucle, todas las instrucciones en el bucle se ejecutan para el primer elemento en *grupo*. Después, mientras haya más elementos en *grupo*, las instrucciones en el bucle continúan ejecutándose para cada elemento. Cuando no hay más elementos en el *grupo*, se sale del bucle y la ejecución continúa con la instrucción que sigue a la instrucción **Next**.

Se pueden colocar en el bucle cualquier número de instrucciones **Exit For**. La instrucción **Exit For** se utiliza a menudo en la evaluación de alguna condición (por ejemplo, **If...Then**) y transfiere el control a la instrucción que sigue inmediatamente a la instrucción **Next**.

Puede anidar bucles **For Each...Next**, colocando un bucle **For Each...Next** dentro de otro. Sin embargo, cada *elemento* del bucle debe ser único.

Nota Si omite *elemento* en una instrucción **Next**, la ejecución continúa como si se hubiera incluido. Si se encuentra una instrucción **Next** antes de su instrucción **For** correspondiente, se producirá un error.

No se puede utilizar la instrucción **For Each...Next** con una matriz de tipos definidos por el usuario porque un tipo **Variant** no puede contener un tipo definido por el usuario.

```
Dim Matriz(10) As Integer
Dim x As Integer
Dim y, suma
suma = 0
   For x = 1 To 10
      Matriz(x) = CInt(InputBox("Dame un numero"))
   Next x
   For Each y In Matriz
      suma = suma + Matriz(y)
   Next y
   MsgBox suma
```

Ejemplo, el siguiente procedimiento **Sub** abre Biblio.mdb y agrega el nombre de cada tabla a un cuadro de lista.

```
Sub ListTableDefs()
   Dim objDb As Database
   Dim MyTableDef as TableDef
   Set objDb = OpenDatabase("c:\vb\biblio.mdb", _
   True, False)
   For Each MyTableDef In objDb.TableDefs()
```

```
        List1.AddItem MyTableDef.Name
    Next MyTableDef
End Sub
```

Tenga en cuenta las restricciones siguientes cuando utilice **For Each...Next**:

- Para las colecciones, *elemento* sólo puede ser una variable **Variant**, una variable **Object** genérica o un objeto mostrado en el Examinador de objetos.
- Para las matrices, *elemento* sólo puede ser una variable **Variant**.
- No puede usar **For Each...Next** con una matriz de tipos definidos por el usuario porque un **Variant** no puede contener un tipo definido por el usuario.

Estructuras de control anidadas

Puede colocar estructuras de control dentro de otras estructuras de control (como un bloque **If...Then** dentro de un bucle **For...Next**). Se dice que una estructura de control colocada dentro de otra estructura de control está *anidada*.

Puede anidar las estructuras de control en Visual Basic en tantos niveles como desee. Es una práctica común crear estructuras de decisión anidadas y estructuras de bucle más legibles sangrando el cuerpo de la estructura de decisión o de bucle.

Cada **Next** cierra el bucle **For** interior y que el último **For** cierra el bucle **For** exterior. Del mismo modo, en instrucciones **If** anidadas, las instrucciones **End If** se aplican automáticamente a la anterior instrucción **If** más cercana. Las estructuras **Do...Loop** anidadas funcionan de una forma parecida; la instrucción **Loop** más interior coincide con la instrucción **Do** más interior.

Matrices de variables

Las matrices le permiten hacer referencia por el mismo nombre a una serie de variables y usar un número (índice) para distinguirlas.

- Todos los elementos de una matriz tienen el mismo tipo de datos (*Por supuesto, cuando el tipo de dato es **Variant**, los elementos individuales pueden contener distintas clases de datos -objetos, cadenas, números, etc.-)*, Puede declarar una matriz de cualquiera de los tipos de datos fundamentales, incluyendo los tipos definidos por el usuario y variables de objetos.
- Las matrices tienen un límite superior e inferior y los elementos de la matriz son contiguos dentro de esos límites.
- Las matrices pueden tener una o más dimensiones.

Esto le ayuda a crear código más pequeño y simple en muchas situaciones, ya que puede establecer *bucles (ciclos)* que traten de forma eficiente cualquier número de casos mediante el número del índice. Puesto que Visual Basic asigna espacio para cada número de índice, evite declarar las matrices más grandes de lo necesario.

Nota Las matrices de variables son declaradas en el código. Son distintas de las matrices de controles que se especifica al establecer la propiedad **Index** de los controles en tiempo de diseño. *Las matrices de variables siempre son contiguas*; a diferencia de las matrices de controles, no puede cargar y descargar elementos de la mitad de la matriz.

En Visual Basic hay dos tipos de matrices:

- las *matrices de tamaño fijo* que tienen siempre el mismo tamaño
- las *matrices dinámicas* cuyo tamaño cambia en tiempo de ejecución.

Declarar matrices de tamaño fijo

A diferencia de otras versiones de Basic, Visual Basic no permite declarar implícitamente una matriz, y tiene que ser declarada explícitamente,

Hay tres formas de declarar una matriz de tamaño fijo, dependiendo del alcance que desee que tenga la matriz:

- Para declarar una *matriz pública*, utilice la instrucción **Public** en la sección Declaraciones de un módulo para declarar la matriz.
- Para crear una *matriz a nivel de módulo*, utilice la instrucción **Private** en la sección Declaraciones de un módulo para declarar la matriz.
- Para crear una *matriz local*, utilice la instrucción **Private** en un procedimiento para declarar la matriz.

Matrices unidimensionales

Cuando declare una matriz **de una dimensión**, ponga a continuación del nombre de la matriz el límite superior entre paréntesis. El límite superior no puede exceder el intervalo de un tipo de dato **Long** (-2.147.483.648 a 2.147.483.647 es decir en el rango -2^{31} a 2^{31}), el límite inferior predeterminado es 0.

Por ejemplo, estas declaraciones de matrices pueden aparecer en la sección Declaraciones de un módulo:

```
Dim Contadores(14) As Integer    ' 15 elementos.
Dim Sumas(20) As Double          ' 21 elementos.
```

La primera declaración crea una matriz de 15 elementos, con números de índice que van de 0 a 14. La segunda crea una matriz de 21 elementos, con números de índice que van de 0 a 20.

Para crear una matriz pública, simplemente utilice **Public** en lugar de **Dim**:

```
Public Contadores(14) As Integer
Public Sumas(20) As Double
```

Las mismas declaraciones dentro de un procedimiento utilizan **Dim**:

```
Dim Contadores(14) As Integer
Dim Sumas(20) As Double
```

Para especificar un límite inferior, proporciónelo explícitamente (como tipo de dato **Long**) mediante la palabra clave **To**:

```
Dim Contadores(1 To 15) As Integer
Dim Sumas(100 To 120) As String
```

En las anteriores declaraciones, los números de índice de `Contadores` van de 1 a 15 y los números de índice de `Sumas` van de 100 a 120.

Matrices multidimensionales

A veces necesitará hacer un seguimiento de la información relacionada de una matriz. Por ejemplo, para hacer un seguimiento de cada píxel de la pantalla del equipo, necesitará referirse a sus coordenadas X e Y. Esto se puede hacer mediante una matriz multidimensional para almacenar los valores.

En Visual Basic puede declarar matrices de varias dimensiones. Por ejemplo, la instrucción siguiente declara una matriz bidimensional de 10 por -10 en un procedimiento:

```
Static MatrizA(9, 9) As Double
```

Se puede declarar una o ambas dimensiones con límites inferiores explícitos:

```
Static MatrizA(1 To 10, 1 To 10) As Double
```

Puede hacer que tenga más de dos dimensiones. Por ejemplo:

```
Dim MultiD(3, 1 To 10, 1 To 15)
```

Esta declaración crea una matriz que tiene tres dimensiones con tamaños de 4 por 10 por 15. El número total de elementos es el producto de las tres dimensiones, es decir, 600.

Nota Cuando comience a agregar dimensiones a una matriz, el almacenamiento total que necesita la matriz se incrementa considerablemente, por lo que debe usar las matrices multidimensionales con sumo cuidado. Tenga especial cuidado con las matrices **Variant**, ya que son más grandes que otros tipos de datos.

Usar bucles para manipular matrices

Puede procesar eficientemente una matriz multidimensional mediante bucles **For** anidados. Por ejemplo, estas instrucciones inicializan cada elemento de `MatrizA` a un valor basándose en su ubicación en la matriz:

```
Dim I As Integer, J As Integer
Static MatrizA(1 To 10, 1 To 10) As Double
```

```
For I = 1 To 10
   For J = 1 To 10
      MatrizA(I, J) = I * 10 + J
   Next J
Next I
```

Matrices dinámicas

A veces necesitará saber exactamente lo grande que debe ser una matriz. Puede que desee poder cambiar el tamaño de la matriz en tiempo de ejecución.

Una matriz dinámica se puede cambiar de tamaño en cualquier momento. Las matrices dinámicas son una de las características más flexibles y cómodas de Visual Basic, y le ayudan a administrar de forma eficiente la memoria. Por ejemplo, puede usar una matriz grande durante un tiempo corto y liberar memoria del sistema cuando no necesite volver a usar la matriz.

La alternativa consiste en declarar la matriz con el mayor tamaño posible y pasar por alto los elementos de la matriz que no necesite. Sin embargo, esta solución, si se utiliza demasiado, puede hacer que el sistema operativo funcione con muy poca memoria.

Para crear una matriz dinámica

1. Declare la matriz con la instrucción **Public** (si desea que la matriz sea pública), la instrucción **Dim** a nivel de módulo (si desea que sea una matriz a nivel de módulo), o con la instrucción **Static** o **Dim** en un procedimiento (si desea que la matriz sea local).
2. Declare la matriz como dinámica proporcionándole una lista de dimensiones vacía.

   ```
   Dim MatrizDyn()
   ```
3. Asigne el número real de elementos con la instrucción **ReDim**.

   ```
   ReDim MatrizDyn(X + 1)
   ```

La instrucción **ReDim** puede aparecer sola en un procedimiento. A diferencia de las instrucciones **Dim** y **Static**, **ReDim** es una instrucción ejecutable; hace que la aplicación realice una acción en tiempo de ejecución.

La instrucción **ReDim** acepta la misma sintaxis que se utiliza en las matrices fijas. Cada **ReDim** puede cambiar el número de elementos, así como los límites inferior y superior de cada dimensión. Sin embargo, no se puede cambiar el número de dimensiones de la matriz.

```
ReDim MatrizDyn(4 to 12)
```

Por ejemplo, la matriz dinámica `Matriz1` se crea declarándola primero a nivel de módulo:

```
Dim Matriz1() As Integer
```

Luego, un procedimiento asigna el espacio de la matriz:

```
Sub CalcValoresAhora ()
   ...
   ReDim Matriz1(19, 29)
End Sub
```

La instrucción **ReDim** mostrada aquí asigna una matriz de 20 por 30 enteros (0 a 19 y 0 a 29, con un tamaño total de 600 elementos). Como alternativa, se pueden establecer los límites de una matriz dinámica mediante variables:

```
ReDim Matriz1(X, Y)
```

Preservar el contenido de las matrices dinámicas

Cada vez que ejecute la instrucción **ReDim** perderá todos los valores almacenados en ese momento en la matriz. Visual Basic restablece los valores al valor **Empty** (en matrices **Variant**), a cero (en matrices numéricas), a una cadena de longitud cero (en matrices de cadenas) o a **Nothing** (en matrices de objetos).

Esto resulta muy útil cuando desee preparar la matriz para contener datos nuevos o cuando desee reducir el tamaño de la matriz para que ocupe menos memoria. Puede que a veces desee cambiar el tamaño de la matriz sin perder los datos de la misma. Para ello puede usar **ReDim** con la palabra clave **Preserve**.

Si utiliza la palabra clave **Preserve** sólo puede cambiar el tamaño de la última dimensión de la matriz y no es posible cambiar el número de dimensiones. Por ejemplo, si la matriz sólo tiene una dimensión, puede cambiar el tamaño de esa dimensión porque es la última y única dimensión. Sin embargo, si la matriz tiene dos o más dimensiones, sólo puede cambiar la dimensión de la última y todavía conservar el contenido de la matriz. El ejemplo siguiente muestra cómo puede aumentar el tamaño de la última dimensión de una matriz dinámica sin borrar ninguno de los datos existentes contenidos en la matriz.

```
ReDim X(10, 10, 10)
. . .
ReDim Preserve X(10, 10, 15)
```

Cuando utiliza el argumento **Preserve** puede cambiar el tamaño de la matriz sólo cambiando el límite superior; cambiar el límite inferior produce un error. De modo parecido, sólo se puede cambiar el límite superior de la última dimensión de una matriz multidimensional cuando se utiliza la palabra clave **Preserve**; si cambia alguna otra dimensión o el límite inferior de la última dimensión, se producirá un error en tiempo de ejecución.

Si hace que una matriz sea más pequeña de lo que era, perderá los datos de los elementos eliminados. Si transfiere una matriz a un procedimiento por referencia, no puede cambiar el tamaño de la matriz dentro del procedimiento.

Precaución La instrucción **ReDim** actúa como una instrucción declarativa si la variable que declara no existe en el nivel de módulo o nivel de procedimiento. Si más tarde crea otra variable con el mismo nombre, incluso con un alcance mayor, **ReDim** hará referencia a la creada más tarde y no causará necesariamente un error de compilación, incluso aunque **Option Explicit** esté en efecto. Para evitar estos conflictos, **ReDim** no se debe utilizar como una instrucción declarativa, sino sólo para cambiar el tamaño de las matrices.

Trabajar con objetos

Cuando crea una aplicación en Visual Basic trabaja con objetos. Puede usar los objetos que proporciona Visual Basic como controles, formularios y objetos de acceso a datos. También puede controlar objetos de otras aplicaciones desde su aplicación de Visual Basic. Puede incluso crear sus propios objetos y definir propiedades y métodos adicionales para ellos.

Controlar objetos mediante sus propiedades

Las propiedades individuales varían ya que puede establecer u obtener sus valores. Se pueden establecer algunas propiedades en tiempo de diseño. Puede usar la ventana Propiedades para establecer el valor de dichas propiedades sin tener que escribir código. Algunas propiedades no están disponibles en tiempo de diseño, por lo que necesitará escribir código para establecer esas propiedades en tiempo de ejecución.

Las propiedades que establece y obtiene en tiempo de ejecución se llaman *propiedades de lectura y escritura*. Las propiedades que sólo puede leer en tiempo de ejecución se llaman *propiedades de sólo lectura*.

Establecer los valores de las propiedades

Establezca el valor de una propiedad cuando desee modificar la apariencia o el comportamiento de un objeto. Por ejemplo, modifique la propiedad **Text** de un cuadro de texto para modificar el contenido del cuadro de texto.

Para establecer el valor de una propiedad, utilice la sintaxis siguiente:

```
objeto.propiedad = expresión
```

Las instrucciones siguientes demuestran cómo se establecen las propiedades:

```
Text1.Top = 200             ' Establece la propiedad Top a 200 twips.
Text1.Visible = True         ' Muestra el cuadro de texto.
Text1.Text = "hola"         ' Muestra 'hola' en el cuadro de texto.
```

Obtener los valores de las propiedades

Obtenga el valor de una propiedad cuando desee encontrar el estado de un objeto antes de que el código realice acciones adicionales (como asignar el valor a otro objeto). Por ejemplo, puede devolver la propiedad **Text** a un control de cuadro de texto para determinar el contenido del cuadro de texto antes de ejecutar código que pueda modificar el valor.

En la mayoría de los casos, para obtener el valor de una propiedad se utiliza la sintaxis siguiente:

```
variable = objeto.propiedad
```

También puede obtener el valor de una propiedad como parte de una expresión más compleja, sin tener que asignar el valor de la propiedad a una variable. En el siguiente código de ejemplo se calcula la propiedad **Top** del nuevo miembro de una matriz de controles como la propiedad **Top** del miembro anterior más 400:

```
Private Sub cmdAdd_Click()
   ' [instrucciones]
   optButton(n).Top = optButton(n-1).Top + 400
   ' [instrucciones]
End Sub
```

Sugerencia Si va a usar el valor de una propiedad más de una vez, el código se ejecutará más rápidamente si almacena el valor en una variable.

Realizar acciones con métodos

Los métodos pueden afectar a los valores de las propiedades. Por ejemplo, en el ejemplo de la radio, el método **EstablecerVolumen** cambia la propiedad **Volumen**. De forma similar, en Visual Basic, los cuadros de lista tienen una propiedad **List** que se puede modificar con los métodos **Clear** y **AddItem**.

Usar métodos en el código

Cuando utiliza un método en el código, la forma en que escribe la instrucción depende de los argumentos que necesite el método y de si el método devuelve o no un valor. Cuando un método no necesita argumentos, escriba el código mediante la sintaxis siguiente:

```
objeto.método
```

En este ejemplo, el método **Refresh** vuelve a dibujar el cuadro de imagen:

```
Picture1.Refresh   ' Obliga a redibujar el control.
```

Algunos métodos, como el método **Refresh**, no necesitan argumentos y no devuelven valores.

Si el método necesita más de un argumento, separe los argumentos mediante comas. Por ejemplo, el método **Circle** utiliza argumentos que especifican la ubicación, el radio y el color de un círculo en un formulario:

```
' Dibuja un círculo azul con un radio de 1200 twips.
Form1.Circle (1600, 1800), 1200, vbBlue
```

Si conserva el valor de devolución de un método, debe poner los argumentos entre paréntesis. Por ejemplo, el método **GetData** devuelve una imagen del Portapapeles:

```
Picture = Clipboard.GetData (vbCFBitmap)
```

Si no hay valor de devolución, los argumentos se ponen sin paréntesis. Por ejemplo, el método **AddItem** no devuelve ningún valor:

```
' Agrega el texto 'sunombre' a un cuadro de lista.
List1.AddItem "sunombre"
```

Usar controles

Validar datos de entrada reteniendo el foco en el control

La propiedad **CausesValidation** se utilize conjuntamente con el evento **Validate** para comprobar, antes de permitir que el usuario se mueva a otro control, que los datos introducidos son válidos. De manera predeterminada la propiedad **CausesValidation** de los controles está establecida a **True**, de esta forma al intentar cambiar el foco del control se ejecutara el evento **Validate**

```
Private Sub Text1_Validate(Cancel As Boolean)
    If Text1.Text = " " Then
       MsgBox ("Falta que introduzcas el radio")
       Cancel = True
    End If
End Sub
```

El argumento **Cancel** cuando está establecido a **True**, hace que el control retenga el foco. El evento Validate es más apropiado para la validación de datos que el evento LostFocus, ya que se produce antes de perder el foco del control.

Casilla de verificación (CheckBox)

Cada casilla de verificación es un control que indica si una opción particular está activada o desactivada. Cada casilla es independiente de las demás, ya que cada una tiene su propio nombre (**Name**).

Para saber si una determinada opción está seleccionada, hay que verificar el valor de su propiedad **Value**. Este puede ser 0, si la casilla aparece sin seleccionar (desmarcada); 1. Si la casilla aparece marcada y 2 si la casilla aparece en gris (inhabilitada).

Puede inhabilitar una casilla de verificación (aparece en gris) poniendo su propiedad **Enable** a valor **False**.

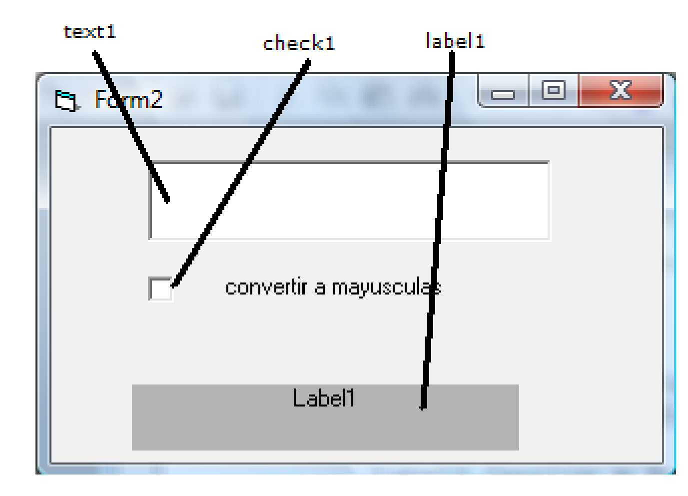
El siguiente código muestra el uso de una caja de texto, de la interfaz:

```
Private Sub Check1_Click()
    Select Case Check1.Value
        Case 0
            Label2.Caption = "convertir a mayusculas"
            Label1.Caption = UCase(Text1.Text)
        Case 1
            Label2.Caption = "convertir a minusculas"
            Label1.Caption = LCase(Text1.Text)
    End Select
End Sub
```

El siguiente procedimiento de evento sirve para realizar la habilitar y deshabilitar la casilla de verificación si el text1 no tiene texto, al verificar el valor de la caja de texto en el momento que cambia el texto en la caja de texto:

```
Private Sub Text1_Change()
```

```
    If Text1.Text = "" Then
      Check1.Enabled = False
    Else
      Check1.Enabled = True
    End If
  End Sub
```



Boton de opción (Radio Button)

Un botón de opción es un control que indica si una determinada opción está activada o desactivada . Cada botón de opción es idependiente de los demás, ya que cada uno de ellos tiene su propio nombre (name). Del número de opciones representas, el usuario solo puede elegir una cada vez.

Para saber si una opción esta seleccionada, hay que verificar el valor de la propiedad **value**. Este valor puede ser falso (***false***) y el botón aparecerá vacío; puede ser verdadero (***true***) y el botón aparecerá seleccionado. Cuando se da click sobre un botón de opción se da el evento Click.

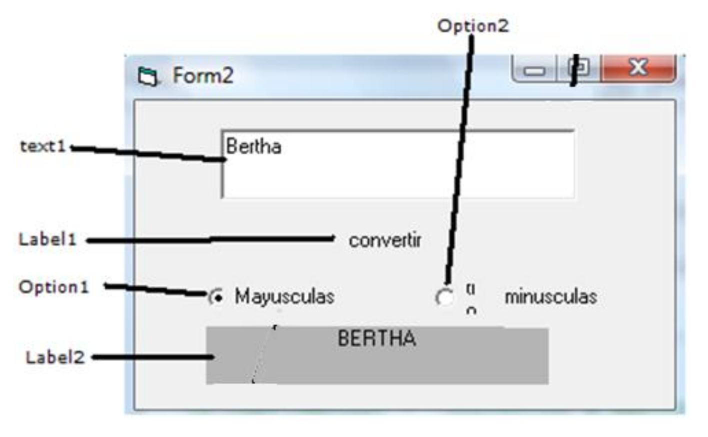
Se inhabilita (aparecerá en gris) o habilita una opción con la propiedad **Enabled** en False o true.

El siguiente código muestra el uso de una caja de texto, de la interfaz:

```
Private Sub Option1_Click()
    Call Estado(Option1.Name)
    Label1.Caption = UCase(Text1.Text)
End Sub

Private Sub Option2_Click()
    Call Estado(Option2.Name)
    Label1.Caption = LCase(Text1.Text)
End Sub

Function Estado(x As String)
    Select Case x
            Case "Option1"
                Label2.Caption = "convertir a mayusculas"
            Case "Option2"
                Label2.Caption = "convertir a minusculas"
    End Select
End Function
```



Listas y Listas desplegables (ListBox y ComboBox)

Una *lista* es un control que pone a disposición del usuario un conjunto de elementos, de los cuales elegirá uno o más. Si la cantidad de elementos rebasa el número de los que pueden ser visualizados simultáneamente en el espacio disponible de la lista, aparecerá automáticamente una barra de desplazamiento vertical para que el usuario pueda desplazar los elementos de la lista hacia arriba o hacia abajo. Por omisión los elementos de una lista son visualizados verticalmente en una sola columna. Se pueden establecer múltiples columnas (en listas simples, no en desplegables) con la propiedad **Column**.

Una lista desplegable es un control que combina las características de una caja de texto y una lista. Esto permite al usuario elegir un elemento de varios, escribiéndolo directamente en la caja de texto o seleccionándolo de la lista.

Con la propiedad **Sorted** en **True** la lista aparecerá clasificada (ordenada)

Para añadir un elemento a una lista o lista desplegable, se usa el método **AddItem**. Sintaxis:

```
Nombre_lista.AddItem elemento[, índice]
```

Donde:

- Nombre_lista es el nombre del control
- Elemento es una cadena de caracteres correspondiente al elemento a añadir.
- Indice, si se especifica, indica la posición donde se insertará el nuevo elemento. Un valor = indica la primera posición. Si no se especifica el índice, el elemento se añade al final de la lista, o bien en la posición que le corresponde si la lista está ordenada.

Para acceder a los elemento de una lista fija o de una lista desplegable, puede utilizar alguna de las propiedades siguientes: ***Text, List, ListIndex,*** *o* ***ListCount****.*

*La propiedad **Text*** para una lista fija o desplegable estilo 2, devuelve el elemento seleccionado en la lista. Para una lista desplegable estilo 0 o 1, devuelve o establece el texto contenido en el área de edición (caja de texto)

```
[formulario!]control.Text[=cadena de caracteres]
```

*La propiedad **List*** se corresponde con una matriz de cadena de caracteres que contiene los elementos de la lista. Por lo tanto, para acceder a uno de estos elementos, simplemente hay que interpretar que List es una matriz. Esto es:

```
[formulario!]control.List(índice)[=cadena de caracteres]
```

El primer elemento tiene como índice 0 y el ultimo, ListCount-1

La propiedad ListCount da como resultado el número de elementos que hay en la lista o lista desplegable. Sintaxis:

```
[formulario!]control.ListCount
```

*La propiedad **ListIndex*** da la posición respecto a 9 del elemento actualmente seleccionado. Si no hay seleccionado ningún elemento, el valor de la propiedad ListIndex es -1. También permite fijar dicha posición.Sintaxis:

```
[formulario!]control.ListIndex[=posición]
```

Para eliminar un elemento de una lista o lista desplegable, es utiliza el método ***RemoveItem***. Sintaxis:

```
Nombre_lista,RemoveItem índice
```

Donde índice indica la posición del elemento que se desea eliminar (0 para la primera posición).

La diferencia entre una lista fija y una lista desplegable (cuadro combinado) es que este último es una combinina una lista y una caja de texto. Las listas desplegables pueden ser de tres estilos:

- CE-1 Una lista desplegable estándar, esta permine que cuando el usuario haga click en la flecha, se visualise la lista de elementos y pueda optar por elegir un elemento de la lista o por escribir directamente el elemento deseado en la caja de texto. Se obtiene poniendo la propiedad ***Style*** en = (Dropdown Combo).
- CE-1 El segundo estilo representa una lista desplegable en la que la lista de elementos siempre está visualizada, el usuario podrá elegir un elemento de la lista o escribir directamente el elemento deseado en la caja de texto. Se obtiene poniendo la propiedad Style en 1 (Simple Combo)
- El tercer estilo representa una lista desplegable en la que el usuario sólo tiene la posibilidad de elegir uno de sus elementos, es decir, no se le permite escribir en la caja de texto. La propiedad Style debe ponerse en 2 (Dropdown List). En otras palabras mas que un cuadro combinado es una lista enrollable.

¿Cómo se relacionan los objetos entre sí?

Cuando pone dos botones de comando en un formulario, son objetos distintos con distinto valor para la propiedad **Name** (*Command1* y *Command2*), pero comparten la misma clase, CommandButton.

También comparten la característica de que están en el mismo formulario. Un control de un formulario también está contenido en el formulario. Esto coloca los controles en una jerarquía. Para hacer referencia a un control debe hacer primero referencia al formulario, de la misma manera que debe marcar el código de un país o el de una provincia antes de poder hablar con un determinado número de teléfono.

Los dos botones de comando comparten también la característica de que son controles. Todos los controles tienen características comunes que los hacen diferentes de los formularios y otros objetos del entorno Visual Basic.

Jerarquías de objetos

La jerarquía de un objeto proporciona la organización que determina cómo se relacionan los objetos entre sí y cómo se puede tener acceso a ellos. En la mayoría de los casos, no es necesario conocer la jerarquía de objetos de Visual Basic. Sin embargo:

- Cuando manipule objetos de otra aplicación, debe estar familiarizado con la jerarquía de objetos de esa aplicación.
- Cuando trabaje con objetos de acceso a datos, debe estar familiarizado con la jerarquía de los Objetos de acceso a datos.

Hay algunos casos comunes en Visual Basic en los que un objeto contiene otros objetos.

Trabajar con colecciones de objetos

Los objetos de una colección tienen sus propias características y métodos. Se hace referencia a los objetos de una colección de objetos como *miembros* de la colección. Cada miembro de una

colección está numerado secuencialmente empezando por 0; éste es el *número de índice* del miembro. Por ejemplo, la colección **Controls** contiene todos los controles de un formulario dado. Puede usar colecciones para simplificar el código si necesita llevar a cabo la misma operación sobre todos los objetos de la colección.

Por ejemplo, el código siguiente se desplaza a través de la colección **Controls** y muestra el nombre de cada miembro en un cuadro de lista.

```
Dim MyControl as Control
For Each MyControl In Form1.Controls
   ' Se agrega el nombre de cada control a un cuadro de lista.
   List1.AddItem MyControl.Name
Next MyControl
```

Aplicar propiedades y métodos a los miembros de una colección

Puede usar dos técnicas generales para dirigirse a un miembro de un objeto colección:

- Especifique el nombre del miembro. Las expresiones siguientes son equivalentes:
- `Controls("List1")`
- `Controls!List1`
- Utilice el número de índice del miembro:
- `Controls(3)`

Una vez que sea capaz de dirigirse a todos los miembros de forma colectiva y a cada miembro de forma individual, puede aplicar propiedades y métodos de la forma siguiente:

```
' Establece la propiedad Top del control de cuadro de lista a 200.
Controls!List1.Top = 200
```

–o bien–

```
Dim MyControl as Control
For Each MyControl In Form1.Controls()
   ' Establece la propiedad Top de cada miembro a 200.
   MyControl.Top = 200
Next MyControl
```

Objetos que contienen otros objetos

Algunos objetos de Visual Basic contienen otros objetos. Por ejemplo, un formulario contiene normalmente uno o más controles. La ventaja de tener objetos como contenedores de otros objetos es que puede hacer referencia al contenedor en el código para que se vea más claramente el objeto que desea usar. Por ejemplo, se podría tener en una aplicación dos formularios distintos; uno para introducir transacciones de cuentas a pagar y otro para introducir transacciones de cuentas a cobrar.

Ambos formularios pueden tener un cuadro de lista llamado lstAcctNo. Puede especificar exactamente cuál desea usar si hace referencia al formulario que contiene el cuadro de lista:

```
frmReceivable.lstAcctNo.AddItem 1201
```

–o bien–

```
frmPayable.lstAcctNo.AddItem 1201
```

Colecciones comunes en Visual Basic

Hay algunos casos comunes en Visual Basic en que un objeto contiene otros objetos. La tabla siguiente describe brevemente las colecciones más utilizadas en Visual Basic.

Colección	**Descripción**
Forms	Contiene los formularios cargados.
Controls	Contiene los controles de un formulario.
Printers	Contiene los objetos **Printer** disponibles.

También puede implementar el contener objetos en Visual Basic.

La propiedad Container

Puede usar la propiedad **Container** para modificar el contenedor de un objeto en un formulario. Los siguientes controles pueden contener otros controles:

- Control Frame
- Control de cuadro Picture
- Control Toolbar (sólo en las ediciones Profesional y Empresarial)

Este ejemplo muestra cómo mover un botón de comando de un contenedor a otro de un formulario. Abra un proyecto nuevo y dibuje en el formulario un control de marco, un control de cuadro con imagen y un botón de comando.

El código siguiente del evento **Click** del formulario incrementa una variable contador y utiliza un bucle **Select Case** para hacer rotar el botón de comando de un contenedor a otro.

```
Private Sub Form_Click()
   Static intX as Integer
   Select Case intX
      Case 0
      Set Command1.Container = Picture1
      Command1.Top= 0
      Command1.Left= 0

      Case 1
      Set Command1.Container = Frame1
      Command1.Top= 0
      Command1.Left= 0

      Case 2
      Set Command1.Container = Form1
      Command1.Top= 0
      Command1.Left= 0

   End Select
   intX = intX + 1
End Sub
```

Anexo 1. Formatos de archivos de proyecto

Microsoft Visual Basic utiliza y crea una serie de archivos tanto en tiempo de diseño como en tiempo de ejecución. Los archivos que el proyecto o la aplicación requerirán dependen de su alcance y funcionalidad.

Extensiones de archivos de proyecto

Visual Basic crea varios archivos cuando se crea y compila un proyecto. Estos se pueden dividir como sigue: tiempo de diseño, otros desarrollos y tiempo de ejecución.

Los archivos de tiempo de diseño son los ladrillos de su proyecto: por ejemplo, módulos de Basic (.bas) y módulos de formulario (.frm).

Otros procesos y funciones del entorno de desarrollo de Visual Basic crean diversos archivos: por ejemplo, archivos de dependencias del Asistente de empaquetado y distribución (.dep).

Archivos varios y de tiempo de diseño

La siguiente tabla muestra todos los archivos de tiempo de diseño y otros archivos que se pueden crear al desarrollar una aplicación:

Extensión	Descripción
.bas	Módulo de Basic
.cls	Módulo de clase
.ctl	Archivo de control de usuario
.ctx	Archivo binario de control de usuario
.dca	Caché de diseñador activo
.ddf	Archivo de información CAB del Asistente de empaquetado y distribución
.dep	Archivo de dependencias del Asistente de empaquetado y distribución
.dob	Archivo de formulario de documento ActiveX
.dox	Archivo binario de formulario de documento ActiveX
.dsr	Archivo de diseñador activo
.dsx	Archivo binario de diseñador activo
.dws	Archivo de secuencia de guiones del Asistente para despliegue
.frm	Archivo de formulario

.frx	Archivo binario de formulario
.log	Archivo de registro de errores de carga
.oca	Archivo de caché de biblioteca de tipos de controles
.pag	Archivo de página de propiedades
.pgx	Archivo binario de página de propiedades
.res	Archivo de recursos
.tlb	Archivo TypeLib de Automatización remota
.vbg	Archivo de proyecto de grupo de Visual Basic
.vbl	Archivo de control de licencia
.vbp	Archivo de proyecto de Visual Basic
.vbr	Archivo de registro de Automatización remota
.vbw	Archivo de espacio de trabajo de proyecto de Visual Basic
.vbz	Archivo de inicio del Asistente
.wct	Plantilla HTML de clase de Web.

Archivos de tiempo de ejecución

Al compilar la aplicación, todos los archivos necesarios de tiempo de ejecución se incluyen en los archivos ejecutables de tiempo de ejecución. La siguiente tabla muestra los archivos de tiempo de ejecución:

Extensión	**Descripción**
.dll	Componente ActiveX en proceso
.exe	Archivo ejecutable o componente ActiveX
.ocx	Control ActiveX
.vbd	Archivo de estado de documento ActiveX
.wct	Plantilla HTML de clase de Web

Anexo 2. Trabajar con formularios MDI y formularios secundarios

Cuando los usuarios de su aplicación MDI abran, guarden y cierren varios formularios secundarios en una sesión, deberían poder hacer referencia al formulario activo y mantener información de estado acerca de los formularios secundarios. En este tema se describen técnicas de codificación que puede usar para especificar el formulario secundario o control activo, cargar y descargar formularios MDI y formularios secundarios, y mantener información de estado acerca de un formulario secundario.

Especificar el formulario secundario o control activo

Algunas veces se necesita proporcionar un comando que opere sobre el control que tenga el enfoque del formulario secundario activo actual. Por ejemplo, suponga que quiere copiar al Portapapeles el texto seleccionado de un cuadro de texto de un formulario secundario.

> Por ejemplo, en una aplicación el evento Click del elemento **Copiar** del menú **Edición** llamará a EditCopyProc, un procedimiento que copia el texto seleccionado al Portapapeles. Como la aplicación puede tener varias instancias del mismo formulario secundario, EditCopyProc necesita saber qué formulario tiene que usar. Para especificarlo, utilice la propiedad **ActiveForm** del formulario MDI, que devuelve el formulario secundario que tiene el enfoque o el que esté activo.
>
> **Nota** Para tener acceso a la propiedad **ActiveForm**, debe estar cargado y visible al menos un formulario MDI secundario; de lo contrario, se producirá un error.

Cuando tenga varios controles en un formulario, también necesita saber cuál es el control activo. Como sucede con la propiedad **ActiveForm**, la propiedad **ActiveControl** devuelve el control del formulario secundario activo que tiene el enfoque. A continuación se ofrece un ejemplo de una subrutina de copia a la que se puede llamar desde un menú de un formulario secundario, desde un menú del formulario MDI o desde un botón de la barra de herramientas:

```
Private Sub EditCopyProc ()
   ' Copia el texto seleccionado al Portapapeles.
   ClipBoard.SetText
      frmMDI.ActiveForm.ActiveControl.SelText
End Sub
```

Si escribe código que se puede llamar desde varias instancias de un formulario, es conveniente *no* usar el identificador del formulario cuando se tenga acceso a los controles o propiedades del formulario. Por ejemplo, haga referencia al alto del cuadro de texto de Form1 con `Text1.Height` en vez de con `Form1.Text1.Height`. De esta manera, el código siempre afectará al formulario actual.

Otra manera de especificar el formulario actual dentro del código es mediante la palabra clave **Me**. **Me** se utiliza para hacer referencia al formulario cuyo código se está ejecutando. Esta palabra clave es útil cuando necesita pasar una referencia a la instancia del formulario actual como argumento de un procedimiento.

Cargar formularios MDI y formularios secundarios

Cuando carga un formulario secundario, se carga y se presenta automáticamente su formulario primario (el formulario MDI). Sin embargo, cuando carga el formulario MDI, sus formularios secundarios no se cargan automáticamente.

En el ejemplo Bloc de notas MDI, el formulario secundario es el formulario inicial predeterminado, por lo que al ejecutar la aplicación se cargan los dos formularios, el MDI y el secundario. Si cambia el formulario inicial de la aplicación Bloc de notas MDI a frmMDI (en la ficha **General** de **Propiedades del proyecto**) y después ejecuta la aplicación, sólo se cargará el formulario MDI. El primer formulario secundario se carga cuando elige **Nuevo** en el menú **Archivo**.

Puede usar la propiedad **AutoShowChildren** para cargar ventanas MDI secundarias como ocultas y mostrarlas después con el método **Show**. Esto le permite actualizar distintos detalles como títulos, posiciones y menús, antes de hacer visible un formulario secundario.

No puede presentar un formulario MDI secundario o el formulario MDI en forma modal (mediante el método **Show** con el argumento **vbModal**). Si desea usar un cuadro de diálogo modal en una aplicación MDI, utilice un formulario con su propiedad **MDIChild** establecida a **False**.

Establecer el tamaño y la posición de un formulario secundario

Cuando un formulario MDI secundario tiene los bordes de tamaño ajustable (**BorderStyle** = 2), Microsoft Windows determina su alto, ancho y posición iniciales al cargarlo. En tiempo de diseño, el tamaño y la posición iniciales de un formulario secundario con bordes de tamaño ajustable depende del tamaño del formulario MDI, no del tamaño del formulario secundario en tiempo de diseño. Cuando los bordes de un formulario MDI secundario no son de tamaño ajustable (**BorderStyle** = 0, 1 ó 3), se carga mediante las propiedades **Height** y **Width** definidas en tiempo de diseño.

Si establece **AutoShowChildren** a **False**, puede modificar la posición del formulario MDI secundario después de cargarlo, pero antes de hacerlo visible.

Mantener información de estado de un formulario secundario

Si un usuario decide salir de la aplicación MDI, debe tener la posibilidad de guardar su trabajo. Para ello, la aplicación necesita poder determinar en todo momento si los datos del formulario secundario han cambiado desde la última vez que se guardaron.

Puede obtener esta información si declara una variable pública dentro de cada formulario secundario. Por ejemplo, puede declarar una variable en la sección Declaraciones de un formulario secundario:

```
Public boolDirty As Boolean
```

Cada vez que el texto de Text1 cambie, el evento Change del cuadro de texto del formulario secundario establece `boolDirty` a **True**. Puede agregar este código para indicar que el contenido de Text1 ha cambiado desde la última vez que se guardó:

```
Private Sub Text1_Change ()
   boolDirty = True
End Sub
```

Por otro lado, cada vez que el usuario guarde el contenido del formulario secundario, el evento Change del cuadro de texto del formulario secundario establece `boolDirty` a **False** para indicar que no se necesita guardar el contenido de Text1. En el siguiente código, se supone que hay un comando de menú llamado **Guardar** (mnuFileSave) y un procedimiento llamado FileSave que guarda el contenido del cuadro de texto:

```
Sub mnuFileSave_Click ()
   ' Guarda el contenido de Text1.
   FileSave
   ' Establece la variable de estado.
   boolDirty = False
End Sub
```

Descargar formularios MDI con QueryUnload

El indicador `boolDirty` es muy útil cuando el usuario decide salir de la aplicación. Esto puede suceder cuando el usuario elige **Cerrar** en el menú **Control** del formulario MDI o mediante un elemento de menú, como **Salir** en el menú **Archivo**. Si el usuario cierra la aplicación desde el menú **Control** del formulario MDI, Visual Basic intentará descargar el formulario MDI.

Cuando se descarga un formulario MDI, se llama al evento QueryUnload primero para el formulario MDI y después para cada formulario secundario que haya abierto. Si el código de ninguno de estos procedimientos de evento QueryUnload cancela el evento Unload, se descargan todos los secundarios y, finalmente, se descarga el formulario MDI.

Puesto que se llama al evento QueryUnload antes de descargar un formulario, puede dar al usuario la oportunidad de guardar un formulario antes de descargarlo. El siguiente código utiliza el indicador `boolDirty` para determinar si se tiene que pedir al usuario que guarde el formulario secundario antes de descargarlo. Observe que puede tener acceso al valor de una variable pública a nivel de formulario desde cualquier parte del proyecto. Este código asume que hay un procedimiento, FileSave, que guarda el contenido de Text1 en un archivo.

```
Private Sub mnuFExit_Click()
   ' Cuando el usuario elige Archivo Salir en una
   ' aplicación MDI, se descarga el formulario MDI
   ' y se llama al evento QueryUnload
   ' para cada secundario abierto.
   Unload frmMDI
   End
End Sub

Private Sub Form_QueryUnload(Cancel As Integer, _
   UnloadMode As Integer)
```

```
   If boolDirty Then
      ' Llama a la rutina que pregunta al usuario y
      ' guarda el archivo si es necesario.
      FileSave
   End If
End Sub
```

Además de los fundamentos de diseño de formularios, tiene que pensar en el inicio y la finalización de la aplicación. Existen varias técnicas para determinar lo que el usuario verá cuando se inicie la aplicación. También es importante prestar atención a los procesos que tienen lugar cuando una aplicación se descarga.

Matrices que contienen otras matrices

Es posible crear una matriz **Variant** y llenarla con otras matrices de distintos tipos de datos. El código siguiente crea dos matrices, una que contiene enteros y otra que contiene cadenas. Se declara entonces una tercera matriz **Variant** y se llena con las matrices de enteros y cadenas.

```
Private Sub Command1_Click()
   Dim intX As Integer   ' Declara la variable de contador.
   ' Declara y llena una matriz de enteros.
   Dim contadoresA(5) As Integer
      For intX = 0 To 4
         contadoresA(intX) = 5
      Next intX
   ' Declara y llena una matriz de cadenas.
      Dim contadoresB(5) As String
         For intX = 0 To 4
            contadoresB(intX) = "adiós"
         Next intX
   Dim arrX(2) As Variant         ' Declara una matriz nueva de dos
                                 ' miembros.
      arrX(1) = contadoresA()    ' Llena la matriz con las otras
                                 ' matrices.
      arrX(2) = contadoresB()
      MsgBox arrX(1)(2)            ' Muestra un miembro de cada
                                 ' matriz.
      MsgBox arrX(2)(3)
End Sub
```

Para obtener más información en general   Vea la Referencia del lenguaje en la ayuda electrónica o en línea de MSDN.